Contents

Architecture Overview

Specification for the Rx RPC Facility

Authentication Server Interface

BOS Server Specification

File Server/Cache Manager Interface

Protection Server Interface

Volume Server/VL Server Interface

Architecture Overview

I

Architecture Overview

Chapter 1

Introduction

1.1 Goals and Background

This paper provides an architectural overview of Transarc's wide-area distributed file system, AFS. Specifically, it covers the current level of available software, the third-generation AFS-3 system. This document will explore the technological climate in which AFS was developed, the nature of problem(s) it addresses, and how its design attacks these problems in order to realize the inherent benefits in such a file system. It also examines a set of additional features for AFS, some of which are actively being considered.

This document is a member of a reference suite providing programming specifications as to the operation of and interfaces offered by the various AFS system components. It is intended to serve as a high-level treatment of distributed file systems in general and of AFS in particular. This document should ideally be read before any of the others in the suite, as it provides the organizational and philosophical framework in which they may best be interpreted.

1.2 Document Layout

Chapter 2 provides a discussion of the technological background and developments that created the environment in which AFS and related systems were inspired. Chapter 3 examines the specific set of goals that AFS was designed to meet, given the possibilities created by personal computing and advances in communication technology. Chapter 4 presents the core AFS architecture and how it addresses these goals. Finally, Chapter 5 considers how AFS functionality may be be improved by certain design changes.

1.3 Related Documents

The names of the other documents in the collection, along with brief summaries of their contents, are listed below.

- *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*: This document describes the *File Server* and *Cache Manager* agents, which provide the backbone file managment services for AFS. The collection of *File Server*s for a cell supplies centralized file storage for that site, and allows clients running the *Cache Manager* component to access those files in a high-performance, secure fashion.
- *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.
- *AFS-3 Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for mapping printable user names to and from their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users, which are suitable for placement on Access Control Lists (ACLs).
- *AFS-3 Programmer's Reference: BOS Server Interface*: This paper covers the "nanny" service which assists in the administrability of the AFS environment.
- *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*: This document specifies the design and operation of the remote procedure call and lightweight process packages used by AFS.

Chapter 2

Technological Background

Certain changes in technology over the past two decades greatly influenced the nature of computational resources, and the manner in which they were used. These developments created the conditions under which the notion of a **distributed file systems (DFS)** was born. This chapter describes these technological changes, and explores how a distributed file system attempts to capitalize on the new computing environment's strengths and minimize its disadvantages.

2.1 Shift in Computational Idioms

By the beginning of the 1980's, new classes of computing engines and new methods by which they may be interconnected were becoming firmly established. At this time, a shift was occurring away from the conventional mainframe-based, timeshared computing environment to one in which both workstation-class machines and the smaller personal computers (PCs) were a strong presence.

The new environment offered many benefits to its users when compared with timesharing. These smaller, self-sufficient machines moved dedicated computing power and cycles directly onto people's desks. Personal machines were powerful enough to support a wide variety of applications, and allowed for a richer, more intuitive, more graphically-based interface for them. Learning curves were greatly reduced, cutting training costs and increasing new-employee productivity. In addition, these machines provided a constant level of service throughout the day. Since a personal machine was typically only executing programs for a single human user, it did not suffer from timesharing's load-based response time degradation. Expanding the computing services for an organization was often accomplished by simply purchasing more of the relatively cheap machines. Even small organizations could now afford their own computing resources, over which they exercised full control. This provided more freedom to tailor computing services to the specific needs of particular groups.

However, many of the benefits offered by the timesharing systems were lost when the computing idiom first shifted to include personal-style machines. One of the prime casualties of this shift was the loss of the notion of a single name space for all files. Instead, workstation- and PC-based environments each had independent and completely disconnected file systems. The standardized mechanisms through which files could be transferred between machines (e.g., `FTP`) were largely designed at a time when there were relatively few large machines that were connected over slow links. Although the newer multi-megabit per second communication pathways allowed for faster transfers, the problem of resource location in this environment was still not addressed. There was no longer a system-wide file system, or even a file location service, so individual users were more isolated from the organization's collective data. Overall, disk requirements ballooned, since lack of a shared file system was often resolved by replicating all programs and data to each machine that needed it. This proliferation of independent copies further complicated the problem of version control and management in this distributed world. Since computers were often no longer behind locked doors at a computer center, user authentication and authorization tasks became more complex. Also, since organizational managers were now in direct control of their computing facilities, they had to also actively manage the hardware and software upon which they depended.

Overall, many of the benefits of the proliferation of independent, personal-style machines were partially offset by the communication and organizational penalties they imposed. Collaborative work and dissemination of information became more difficult now that the previously unified file system was fragmented among hundreds of autonomous machines.

2.2 Distributed File Systems

As a response to the situation outlined above, the notion of a **distributed file system (DFS)** was developed. Basically, a DFS provides a framework in which access to files is permitted regardless of their locations. Specifically, a distributed file system offers a single, common set of file system operations through which those accesses are performed.

There are two major variations on the core DFS concept, classified according to the way in which file storage is managed. These high-level models are defined below.

- **Peer-to-peer**: In this symmetrical model, each participating machine provides storage for specific set of files on its own attached disk(s), and allows others to access them remotely. Thus, each node in the DFS is capable of both *importing* files (making reference to files resident on foreign machines) and *exporting* files (allowing other machines to reference files located locally).

- **Server-client**: In this model, a set of machines designated as *servers* provide the storage for all of the files in the DFS. All other machines, known as *clients*, must direct their file references to these machines. Thus, servers are the sole exporters of files in the DFS, and clients are the sole importers.

The notion of a DFS, whether organized using the peer-to-peer or server-client discipline, may be used as a conceptual base upon which the advantages of personal computing resources can be combined with the single-system benefits of classical timeshared operation.

Many distributed file systems have been designed and deployed, operating on the fast local area networks available to connect machines within a single site. These systems include DOMAIN [9], DS [15], RFS [16], and Sprite [10]. Perhaps the most widespread of distributed file systems to date is a product from Sun Microsystems, NFS [4] [5], extending the popular UNIX file system so that it operates over local networks.

2.3 Wide-Area Distributed File Systems

Improvements in long-haul network technology are allowing for faster interconnection bandwidths and smaller latencies between distant sites. Backbone services have been set up across the country, and T1 (1.5 megabit/second) links are increasingly available to a larger number of locations. Long-distance channels are still at best approximately an order of magnitude slower than the typical local area network, and often two orders of magnitude slower. The narrowed *difference* between local-area and wide-area data paths opens the window for the notion of a **wide-area distributed file system (WADFS)**. In a WADFS, the transparency of file access offered by a local-area DFS is extended to cover machines across much larger distances. Wide-area file system functionality facilitates collaborative work and dissemination of information in this larger theater of operation.

Chapter 3

AFS-3 Design Goals

3.1 Introduction

This chapter describes the goals for the AFS-3 system, the first commercial WADFS in existence.

The original AFS goals have been extended over the history of the project. The initial AFS concept was intended to provide a single distributed file system facility capable of supporting the computing needs of Carnegie Mellon University, a community of roughly 10,000 people. It was expected that most CMU users either had their own workstation-class machine on which to work, or had access to such machines located in public *clusters*. After being successfully implemented, deployed, and tuned in this capacity, it was recognized that the basic design could be augmented to link autonomous AFS installations located within the greater CMU campus. As described in Section 2.3, the long-haul networking environment developed to a point where it was feasible to further extend AFS so that it provided wide-area file service. The underlying AFS communication component was adapted to better handle the widely-varying channel characteristics encountered by intra-site and inter-site operations.

A more detailed history of AFS evolution may be found in [3] and [18].

3.2 System Goals

At a high level, the AFS designers chose to extend the single-machine UNIX computing environment into a WADFS service. The UNIX system, in all of its numerous incarnations, is an important computing standard, and is in very wide use. Since AFS was originally intended to service the heavily UNIX-oriented CMU campus, this decision served an important tactical purpose along with its strategic ramifications.

In addition, the server-client discipline described in Section 2.2 was chosen as the organizational base for AFS. This provides the notion of a central file store serving as the primary residence for files within a given organization. These centrally-stored files are maintained by server machines and are made accessible to computers running the AFS client software.

Listed in the following sections are the primary goals for the AFS system. Chapter 4 examines how the AFS design decisions, concepts, and implementation meet this list of goals.

3.2.1 Scale

AFS differs from other existing DFSs in that it has the specific goal of supporting a very large user community with a small number of server machines. Unlike the rule-of-thumb ratio of approximately 20 client machines for every server machine (20:1) used by Sun Microsystem's widespread NFS distributed file system, the AFS architecture aims at smoothly supporting client/server ratios more along the lines of 200:1 within a single installation. In addition to providing a DFS covering a single organization with tens of thousands of users, AFS also aims at allowing thousands of independent, autonomous organizations to join in the single, shared name space (see Section 3.2.2 below) without a centralized control or coordination point. Thus, AFS envisions supporting the file system needs of tens of millions of users at interconnected yet autonomous sites.

3.2.2 Name Space

One of the primary strengths of the timesharing computing environment is the fact that it implements a single name space for all files in the system. Users can walk up to any terminal connected to a timesharing service and refer to its files by the identical name. This greatly encourages collaborative work and dissemination of information, as everyone has a common frame of reference. One of the major AFS goals is the extension of this concept to a WADFS. Users should be able to walk up to any machine acting as an AFS client, anywhere in the world, and use the identical file name to refer to a given object.

In addition to the common name space, it was also an explicit goal for AFS to provide complete **access transparency** and **location transparency** for its files. *Access* *transparency* is defined as the system's ability to use a single mechanism to operate on a file, regardless of its location, local or remote. *Location transparency* is defined as the inability to determine a file's location from its name. A system offering location transparency may also provide transparent file mobility, relocating files between server machines without visible effect to the naming system.

3.2.3 Performance

Good system performance is a critical AFS goal, especially given the scale, client-server ratio, and connectivity specifications described above. The AFS architecture aims at providing file access characteristics which, on average, are similar to those of local disk performance.

3.2.4 Security

A production WADFS, especially one which allows and encourages transparent file access between different administrative domains, must be extremely conscious of security issues. AFS assumes that server machines are "trusted" within their own administrative domain, being kept behind locked doors and only directly manipulated by reliable administrative personnel. On the other hand, AFS client machines are assumed to exist in inherently insecure environments, such as offices and dorm rooms. These client machines are recognized to be unsupervisable, and fully accessible to their users. This situation makes AFS servers open to attacks mounted by possibly modified client hardware, firmware, operating systems, and application software. In addition, while an organization may actively enforce the physical security of its own file servers to its satisfaction, other organizations may be lax in comparison. It is important to partition the system's security mechanism so that a security breach in one administrative domain does not allow unauthorized access to the facilities of other autonomous domains.

The AFS system is targeted to provide confidence in the ability to protect system data from unauthorized access in the above environment, where untrusted client hardware and software may attempt to perform direct remote file operations from anywhere in the world, and where levels of physical security at remote sites may not meet the standards of other sites.

3.2.5 Access Control

The standard UNIX access control mechanism associates *mode bits* with every file and directory, applying them based on the user's numerical identifier and the user's membership in various groups. This mechanism was considered too coarse-grained by the AFS designers. It was seen as insufficient for specifying the exact set of individuals and groups which may properly access any given file, as well as the operations these principals may perform. The UNIX group mechanism was also considered too coarse and inflexible. AFS was designed to provide more flexible and finer-grained control of file access, improving the ability to define the set of parties which may operate on files, and what their specific access rights are.

3.2.6 Reliability

The crash of a server machine in any distributed file system causes the information it hosts to become unavailable to the user community. The same effect is observed when server and client machines are isolated across a network partition. Given the potential size of the AFS user community, a single server crash could potentially deny service to a very large number of people. The AFS design reflects a desire to minimize the visibility and impact of these inevitable server crashes.

3.2.7 Administrability

Driven once again by the projected scale of AFS operation, one of the system's goals is to offer easy administrability. With the large projected user population, the amount of file data expected to be resident in the shared file store, and the number of machines in the environment, a WADFS could easily become impossible to administer unless its design allowed for easy monitoring and manipulation of system resources. It is also imperative to be able to apply security and access control mechanisms to the administrative interface.

3.2.8 Interoperability/Coexistence

Many organizations currently employ other distributed file systems, most notably Sun Microsystem's NFS, which is also an extension of the basic single-machine UNIX system. It is unlikely that AFS will receive significant use if it cannot operate concurrently with other DFSs without mutual interference. Thus, coexistence with other DFSs is an explicit AFS goal.

A related goal is to provide a way for other DFSs to interoperate with AFS to various degrees, allowing AFS file operations to be executed from these competing systems. This is advantageous, since it may extend the set of machines which are capable of interacting with the AFS community. Hardware platforms and/or operating systems to which AFS is not ported may thus be able to use their native DFS system to perform AFS file references.

These two goals serve to extend AFS coverage, and to provide a migration path by which potential clients may sample AFS capabilities, and gain experience with AFS. This may result in data migration into native AFS systems, or the impetus to acquire a native AFS implementation.

3.2.9 Heterogeneity/Portability

It is important for AFS to operate on a large number of hardware platforms and operating systems, since a large community of unrelated organizations will most likely utilize a wide variety of computing environments. The size of the potential AFS user community will be unduly restricted if AFS executes on a small number of platforms. Not only must AFS support a largely heterogeneous computing base, it must also be designed to be easily portable to new hardware and software releases in order to maintain this coverage over time.

Chapter 4

AFS High-Level Design

4.1 Introduction

This chapter presents an overview of the system architecture for the AFS-3 WADFS. Different treatments of the AFS system may be found in several documents, including [3], [4], [5], and [2]. Certain system features discussed here are examined in more detail in the set of accompanying AFS programmer specification documents.

After the archtectural overview, the system goals enumerated in Chapter 3 are revisited, and the contribution of the various AFS design decisions and resulting features is noted.

4.2 The AFS System Architecture

4.2.1 Basic Organization

As stated in Section 3.2, a server-client organization was chosen for the AFS system. A group of trusted server machines provides the primary disk space for the central store managed by the organization controlling the servers. File system operation requests for specific files and directories arrive at server machines from machines running the AFS client software. If the client is authorized to perform the operation, then the server proceeds to execute it.

In addition to this basic file access functionality, AFS server machines also provide related system services. These include authentication service, mapping between printable and numerical user identifiers, file location service, time service, and such administrative operations as disk management, system reconfiguration, and tape backup.

4.2.2 Volumes

4.2.2.1 Definition

Disk partitions used for AFS storage do not directly host individual user files and directories. Rather, connected subtrees of the system's directory structure are placed into containers called **volumes**. Volumes vary in size dynamically as the objects it houses are inserted, overwritten, and deleted. Each volume has an associated **quota**, or maximum permissible storage. A single UNIX disk partition may thus host one or more volumes, and in fact may host as many volumes as physically fit in the storage space. However, the practical maximum is currently 3,500 volumes per disk partition. This limitation is imposed by the *salvager* program, which examines and repairs file system metadata structures.

There are two ways to identify an AFS volume. The first option is a 32-bit numerical value called the **volume ID**. The second is a human-readable character string called the **volume name**.

Internally, a volume is organized as an array of mutable objects, representing individual files and directories. The file system object associated with each index in this internal array is assigned a **uniquifier** and a **data version number**. A subset of these values are used to compose an AFS file identifier, or **FID**. FIDs are not normally visible to user applications, but rather are used internally by AFS. They consist of ordered triplets, whose components are the volume ID, the index within the volume, and the uniquifier for the index.

To understand AFS FIDs, let us consider the case where index i in volume v refers to a file named *example.txt*. This file's uniquifier is currently set to one (1), and its data version number is currently set to zero (0). The AFS client software may then refer to this file with the following FID: (v, i, 1). The next time a client overwrites the object identified with the (v, i, 1) FID, the data version number for *example.txt* will be promoted to one (1). Thus, the data version number serves to distinguish between different versions of the same file. A higher data version number indicates a newer version of the file.

Consider the result of deleting file (v, i, 1). This causes the body of *example.txt* to be discarded, and marks index i in volume v as unused. Should another program create a file, say *a.out*, within this volume, index i may be reused. If it is, the creation operation will bump the index's uniquifier to two (2), and the data version number is reset to zero

(0). Any client caching a FID for the deleted *example.txt* file thus cannot affect the completely unrelated *a.out* file, since the uniquifiers differ.

4.2.2.2 Attachment

The connected subtrees contained within individual volumes are attached to their proper places in the file space defined by a site, forming a single, apparently seamless UNIX tree. These attachment points are called mount points. These mount points are persistent file system objects, implemented as symbolic links whose contents obey a stylized format. Thus, AFS mount points differ from NFS-style *mounts*. In the NFS environment, the user dynamically mounts entire remote disk partitions using any desired name. These mounts do not survive client restarts, and do not insure a uniform namespace between different machines.

A single volume is chosen as the root of the AFS file space for a given organization. By convention, this volume is named `root.afs`. Each client machine belonging to this organization peforms a UNIX *mount()* of this root volume (not to be confused with an AFS mount point) on its empty */afs* directory, thus attaching the entire AFS name space at this point.

4.2.2.3 Administrative Uses

Volumes serve as the administrative unit for AFS file system data, providing as the basis for **replication**, **relocation**, and **backup** operations.

4.2.2.4 Replication

Read-only snapshots of AFS volumes may be created by administrative personnel. These *clones* may be deployed on up to eight disk partitions, on the same server machine or across different servers. Each clone has the identical volume ID, which must differ from its read-write parent. Thus, at most one clone of any given volume v may reside on a given disk partition. File references to this read-only clone volume may be serviced by *any* of the servers which host a copy.

4.2.2.5 Backup

Volumes serve as the unit of tape backup and restore operations. Backups are accomplished by first creating an on-line *backup volume* for each volume to be archived. This backup volume is organized as a *copy-on-write shadow* of the original volume, capturing the volume's state at the instant that the backup took place. Thus, the backup volume may be envisioned as being composed of a set of object pointers back to the original image. The first update operation on the file located in index i of the original volume triggers the copy-on-write association. This causes the file's contents at the time of the snapshot to be physically written to the backup volume before the newer version of the file is stored in the parent volume.

Thus, AFS on-line backup volumes typically consume little disk space. On average, they are composed mostly of links and to a lesser extent the bodies of those few files which have been modified since the last backup took place. Also, the system does not have to be shut down to insure the integrity of the backup images. Dumps are generated from the unchanging backup volumes, and are transferred to tape at any convenient time before the next backup snapshot is performed.

4.2.2.6 Relocation

Volumes may be moved transparently between disk partitions on a given file server, or between different file server machines. The transparency of volume motion comes from the fact that neither the user-visible names for the files nor the internal AFS FIDs contain server-specific location information.

Interruption to file service while a volume move is being executed is typically on the order of a few seconds, regardless of the amount of data contained within the volume. This derives from the staged algorithm used to move a volume to a new server. First, a dump is taken of the volume's contents, and this image is installed at the new site. The second stage involves actually locking the original volume, taking an incremental dump to capture file updates since the first stage. The third stage installs the changes at the new site, and the fourth stage deletes the original volume. Further references to this volume will resolve to its new location.

4.2.3 Authentication

AFS uses the Kerberos [6] [7] authentication system developed at MIT's Project Athena to provide reliable identification of the principals attempting to operate on the files in its central store. Kerberos provides for **mutual authentication**, not only assuring AFS servers that they are interacting with the stated user, but also assuring AFS clients that they are dealing with the proper server entities and not imposters. Authentication information is mediated through the use of *tickets*. Clients register passwords with the authentication system, and use those passwords during authentication sessions to secure these tickets. A ticket is an object which contains an encrypted version of the user's name and other information. The file server machines may request a caller to present their ticket in the course of a file system operation. If the file server can successfully decrypt the ticket, then it knows that it was created and delivered by the authentication system, and may trust that the caller is the party identified within the ticket.

Such subjects as mutual authentication, encryption and decryption, and the use of session keys are complex ones. Readers are directed to the above references for a complete treatment of Kerberos-based authentication.

4.2.4 Authorization

4.2.4.1 Access Control Lists

AFS implements per-directory **Access Control Lists (ACLs)** to improve the ability to specify which sets of users have access to the files within the directory, and which operations they may perform. ACLs are used *in addition* to the standard UNIX mode bits. ACLs are organized as lists of one or more *(principal, rights)* pairs. A *principal* may be either the name of an individual user or a group of individual users. There are seven expressible rights, as listed below.

- **Read (r)**: The ability to read the contents of the files in a directory.
- **Lookup (l)**: The ability to look up names in a directory.
- **Write (w)**: The ability to create new files and overwrite the contents of existing files in a directory.
- **Insert (i)**: The ability to insert new files in a directory, but *not* to overwrite existing files.
- **Delete (d)**: The ability to delete files in a directory.
- **Lock (k)**: The ability to acquire and release advisory locks on a given directory.
- **Administer (a)**: The ability to change a directory's ACL.

4.2.4.2 AFS Groups

AFS users may create a certain number of *groups*, differing from the standard UNIX notion of group. These AFS groups are objects that may be placed on ACLs, and simply contain a list of AFS user names that are to be treated identically for authorization purposes. For example, user `erz` may create a group called `erz:friends` consisting of the `kazar`, `vasilis`, and `mason` users. Should `erz` wish to grant read, lookup, and insert rights to this group in directory *d*, he should create an entry reading *(erz:friends, rli)* in *d*'s ACL.

AFS offers three special, built-in groups, as described below.

1. **system:anyuser**: Any individual who accesses AFS files is considered by the system to be a member of this group, whether or not they hold an authentication ticket. This group is unusual in that it doesn't have a stable membership. In fact, it doesn't have an explicit list of members. Instead, the `system:anyuser` "membership" grows and shrinks as file accesses occur, with users being (conceptually) added and deleted automatically as they interact with the system.

   The `system:anyuser` group is typically put on the ACL of those directories for which some specific level of completely public access is desired, covering any user at any AFS site.

2. **system:authuser**: Any individual in possession of a valid Kerberos ticket minted by the organization's authentication service is treated as a member of this group. Just as with `system:anyuser`, this special group does not have a stable membership. If a user acquires a ticket from the authentication service, they are automatically "added" to the group. If the ticket expires or is discarded by the user, then the given individual will automatically be "removed" from the group.

   The `system:authuser` group is usually put on the ACL of those directories for which some specific level of intra-site access is desired. Anyone holding a valid ticket within the organization will be allowed to perform the set of accesses specified by the ACL entry, regardless of their precise individual ID.

3. **system:administrators**: This built-in group defines the set of users capable of performing certain important administrative operations within the cell. Members of this group have explicit "a" (ACL administration) rights on every directory's ACL in the organization. Members of this group are the only ones which may legally issue administrative commands to the file server machines within the organization. This group is *not* like the other two described above in that it *does* have a stable membership, where individuals are added and deleted from the group explicitly.

The `system:administrators` group is typically put on the ACL of those directories which contain sensitive administrative information, or on those places where only administrators are allowed to make changes. All members of this group have implicit rights to change the ACL on any AFS directory within their organization. Thus, they don't have to actually appear on an ACL, or have "a" rights enabled in their ACL entry if they do appear, to be able to modify the ACL.

4.2.5 Cells

A cell is the set of server and client machines managed and operated by an administratively independent organization, as fully described in the original proposal [17] and specification [18] documents. The cell's administrators make decisions concerning such issues as server deployment and configuration, user backup schedules, and replication strategies on their own hardware and disk storage completely independently from those implemented by other cell administrators regarding their own domains. Every client machine belongs to exactly one cell, and uses that information to determine where to obtain default system resources and services.

The cell concept allows autonomous sites to retain full administrative control over their facilities while allowing them to collaborate in the establishment of a single, common name space composed of the union of their individual name spaces. By convention, any file name beginning with */afs* is part of this shared global name space and can be used at any AFS-capable machine. The original mount point concept was modified to contain cell information, allowing volumes housed in foreign cells to be mounted in the file space. Again by convention, the top-level */afs* directory contains a mount point to the `root.cell` volume for each cell in the AFS community, attaching their individual file spaces. Thus, the top of the data tree managed by cell `xyz` is represented by the */afs/xyz* directory.

Creating a new AFS cell is straightforward, with the operation taking three basic steps:

1. **Name selection**: A prospective site has to first select a unique name for itself. Cell name selection is inspired by the hierarchical Domain naming system. Domain-style names are designed to be assignable in a completely decentralized fashion. Example cell names are `transarc.com`, `ssc.gov`, and `umich.edu`. These names correspond to the AFS installations at Transarc Corporation in Pittsburgh, PA, the Superconducting Supercollider Lab in Dallas, TX, and the University of Michigan at Ann Arbor, MI. respectively.

2. **Server installation**: Once a cell name has been chosen, the site must bring up one or more AFS file server machines, creating a local file space and a suite of local

services, including authentication (Section 4.2.6.4) and volume location (Section 4.2.6.2).

3. **Advertise services**: In order for other cells to discover the presence of the new site, it must advertise its name and which of its machines provide basic AFS services such as authentication and volume location. An established site may then record the machines providing AFS system services for the new cell, and then set up its mount point under */afs*. By convention, each cell places the top of its file tree in a volume named `root.cell`.

4.2.6 Implementation of Server Functionality

AFS server functionality is implemented by a set of user-level processes which execute on server machines. This section examines the role of each of these processes.

4.2.6.1 *File Server*

This AFS entity is responsible for providing a central disk repository for a particular set of files within volumes, and for making these files accessible to properly-authorized users running on client machines.

4.2.6.2 *Volume Location Server*

The *Volume Location Server* maintains and exports the **Volume Location Database (VLDB)**. This database tracks the server or set of servers on which volume instances reside. Among the operations it supports are queries returning volume location and status information, volume ID management, and creation, deletion, and modification of VLDB entries.

The VLDB may be replicated to two or more server machines for availability and load-sharing reasons. A *Volume Location Server* process executes on each server machine on which a copy of the VLDB resides, managing that copy.

4.2.6.3 *Volume Server*

The *Volume Server* allows administrative tasks and probes to be performed on the set of AFS volumes residing on the machine on which it is running. These operations include volume creation and deletion, renaming volumes, dumping and restoring volumes, altering the list of replication sites for a read-only volume, creating and propagating a new read-only volume image, creation and update of backup volumes, listing all volumes on a partition, and examining volume status.

4.2.6.4 *Authentication Server*

The AFS *Authentication Server* maintains and exports the **Authentication Database (ADB)**. This database tracks the encrypted passwords of the cell's users. The *Authentication Server* interface allows operations that manipulate ADB entries. It also implements the Kerberos mutual authentication protocol, supplying the appropriate identification tickets to successful callers.

The ADB may be replicated to two or more server machines for availability and load-sharing reasons. An *Authentication Server* process executes on each server machine on which a copy of the ADB resides, managing that copy.

4.2.6.5 *Protection Server*

The *Protection Server* maintains and exports the **Protection Database (PDB)**, which maps between printable user and group names and their internal numerical AFS identifiers. The *Protection Server* also allows callers to create, destroy, query ownership and membership, and generally manipulate AFS user and group records.

The PDB may be replicated to two or more server machines for availability and load-sharing reasons. A *Protection Server* process executes on each server machine on which a copy of the PDB resides, managing that copy.

4.2.6.6 *BOS Server*

The *BOS Server* is an administrative tool which runs on each file server machine in a cell. This server is responsible for monitoring the health of the AFS agent processess on that machine. The *BOS Server* brings up the chosen set of AFS agents in the proper order after a system reboot, answers requests as to their status, and restarts them when they fail. It also accepts commands to start, suspend, or resume these processes, and install new server binaries.

4.2.6.7 *Update Server/Client*

The *Update Server* and *Update Client* programs are used to distribute important system files and server binaries. For example, consider the case of distributing a new *File Server* binary to the set of Sparcstation server machines in a cell. One of the Sparcstation servers is declared to be the distribution point for its machine class, and is configured to run an *Update Server*. The new binary is installed in the appropriate local directory on that Sparcstation distribution point. Each of the other Sparcstation servers runs an *Update Client* instance, which periodically polls the proper *Update Server*. The new *File Server* binary will be detected and copied over to the client. Thus, new server binaries need only be installed manually once per machine type, and the distribution to like server machines will occur automatically.

4.2.7 Implementation of Client Functionality

4.2.7.1 Introduction

The portion of the AFS WADFS which runs on each client machine is called the *Cache Manager*. This code, running within the client's kernel, is a user's representative in communicating and interacting with the *File Server*s. The *Cache Manager*'s primary responsibility is to create the illusion that the remote AFS file store resides on the client machine's local disk(s).

As implied by its name, the *Cache Manager* supports this illusion by maintaining a *cache* of files referenced from the central AFS store on the machine's local disk. All file operations executed by client application programs on files within the AFS name space are handled by the *Cache Manager* and are realized on these cached images. Client-side AFS references are directed to the *Cache Manager* via the standard **VFS** and **vnode** file system interfaces pioneered and advanced by Sun Microsystems [8]. The *Cache Manager* stores and fetches files to and from the shared AFS repository as necessary to satisfy these operations. It is responsible for parsing UNIX pathnames on *open()* operations and mapping each component of the name to the *File Server* or group of *File Server*s that house the matching directory or file.

The *Cache Manager* has additional responsibilities. It also serves as a reliable repository for the user's authentication information, holding on to their tickets and wielding them as necessary when challenged during *File Server* interactions. It caches volume location information gathered from probes to the VLDB, and keeps the client machine's local clock synchronized with a reliable time source.

4.2.7.2 Chunked Access

In previous AFS incarnations, whole-file caching was performed. Whenever an AFS file was referenced, the entire contents of the file were stored on the client's local disk. This approach had several disadvantages. One problem was that no file larger than the amount of disk space allocated to the client's local cache could be accessed.

AFS-3 supports *chunked* file access, allowing individual 64 kilobyte pieces to be fetched and stored. Chunking allows AFS files of any size to be accessed from a client. The chunk size is settable at each client machine, but the default chunk size of 64K was chosen so that most UNIX files would fit within a single chunk.

4.2.7.3 Cache Management

The use of a file cache by the AFS client-side code, as described above, raises the thorny issue of cache consistency. Each client must efficiently determine whether its cached file chunks are identical to the corresponding sections of the file as stored at the server machine before allowing a user to operate on those chunks.

AFS employs the notion of a **callback** as the backbone of its cache consistency algorithm. When a server machine delivers one or more chunks of a file to a client, it also includes a callback "promise" that the client will be notified if any modifications are made to the data in the file at the server. Thus, as long as the client machine is in possession of a callback for a file, it knows it is correctly synchronized with the centrally-stored version, and allows its users to operate on it as desired without any further interaction with the server. Before a file server stores a more recent version of a file on its own disks, it will first break all outstanding callbacks on this item. A callback will eventually time out, even if there are no changes to the file or directory it covers.

4.2.8 Communication Substrate: *Rx*

All AFS system agents employ **remote procedure call (RPC)** interfaces. Thus, servers may be queried and operated upon regardless of their location.

The *Rx* RPC package is used by all AFS agents to provide a high-performance, multi-threaded, and secure communication mechanism. The *Rx* protocol is adaptive, conforming itself to widely varying network communication media encountered by a WADFS. It allows user applications to define and insert their own security modules, allowing them to execute the precise end-to-end authentication algorithms required to suit their specific needs and goals. *Rx* offers two built-in security modules. The first is the *null* module, which does not perform any encryption or authentication checks. The second built-in security module is *rxkad*, which utilizes Kerberos authentication.

Although pervasive throughout the AFS distributed file system, all of its agents, and many of its standard application programs, *Rx* is entirely separable from AFS and does not depend on any of its features. In fact, *Rx* can be used to build applications engaging in RPC-style communication under a variety of UNIX-style file systems. There are in-kernel and user-space implementations of the *Rx* facility, with both sharing the same interface.

4.2.9 Database Replication: *ubik*

The three AFS system databases (VLDB, ADB, and PDB) may be replicated to multiple server machines to improve their availability and share access loads among the replication sites. The *ubik* replication package is used to implement this functionality. A full description of *ubik* and of the *quorum completion* algorithm it implements may be found in [4] and [5].

The basic abstraction provided by *ubik* is that of a disk file replicated to multiple server locations. One machine is considered to be the *synchronization site*, handling all write operations on the database file. Read operations may be directed to any of the active members of the *quorum*, namely a subset of the replication sites large enough to insure integrity across such failures as individual server crashes and network partitions. All of the quorum members participate in regular *elections* to determine the current synchronization site. The *ubik* algorithms allow server machines to enter and exit the quorum in an orderly and consistent fashion.

All operations to one of these replicated "abstract files" are performed as part of a *transaction*. If all the related operations performed under a transaction are successful, then the transaction is *committed*, and the changes are made permanent. Otherwise, the transaction is *aborted*, and all of the operations for that transaction are undone.

Like *Rx*, the *ubik* facility may be used by client applications directly. Thus, user applicatons may easily implement the notion of a replicated disk file in this fashion.

4.2.10 System Management

There are several AFS features aimed at facilitating system management. Some of these features have already been mentioned, such as volumes, the *BOS Server*, and the pervasive use of secure RPCs throughout the system to perform administrative operations from any AFS client machinein the worldwide community. This section covers additional AFS features and tools that assist in making the system easier to manage.

4.2.10.1 Intelligent Access Programs

A set of intelligent user-level applications were written so that the AFS system agents could be more easily queried and controlled. These programs accept user input, then translate the caller's instructions into the proper RPCs to the responsible AFS system agents, in the proper order.

An example of this class of AFS application programs is `vos`, which mediates access to the *Volume Server* and the *Volume Location Server* agents. Consider the `vos move` operation, which results in a given volume being moved from one site to another. The *Volume Server* does not support a complex operation like a volume move directly. In fact, this move operation involves the *Volume Server*s at the current and new machines, as well as the *Volume Location Server*, which tracks volume locations. Volume moves are accomplished by a combination of full and incremental volume dump and restore operations, and a VLDB update. The `vos move` command issues the necessary RPCs in the proper order, and attempts to recovers from errors at each of the steps.

The end result is that the AFS interface presented to system administrators is much simpler and more powerful than that offered by the raw RPC interfaces themselves. The learning curve for administrative personnel is thus flattened. Also, automatic execution of complex system operations are more likely to be successful, free from human error.

4.2.10.2 Monitoring Interfaces

The various AFS agent RPC interfaces provide calls which allow for the collection of system status and performance data. This data may be displayed by such programs as `scout`, which graphically depicts *File Server* performance numbers and disk utilizations. Such monitoring capabilites allow for quick detection of system problems. They also support detailed performance analyses, which may indicate the need to reconfigure system resources.

4.2.10.3 Backup System

A special backup system has been designed and implemented for AFS, as described in [6]. It is not sufficient to simply dump the contents of all *File Server* partitions onto tape, since volumes are mobile, and need to be tracked individually. The AFS backup system allows hierarchical **dump schedules** to be built based on volume names. It generates the appropriate RPCs to create the required backup volumes and to dump these snapshots to tape. A database is used to track the backup status of system volumes, along with the set of tapes on which backups reside.

4.2.11 Interoperability

Since the client portion of the AFS software is implemented as a standard VFS/vnode file system object, AFS can be installed into client kernels and utilized without interference with other VFS-style file systems, such as vanilla UNIX and the NFS distributed file system.

Certain machines either cannot or choose not to run the AFS client software natively. If these machines run NFS, it is still possible to access AFS files through a *protocol translator*. The **NFS-AFS Translator** may be run on any machine at the given site that runs both NFS and the AFS *Cache Manager*. All of the NFS machines that wish to access the AFS shared store proceed to NFS-mount the translator's */afs* directory. File references generated at the NFS-based machines are received at the translator machine, which is acting in its capacity as an NFS server. The file data is actually obtained when the translator machine issues the corresponding AFS references in its role as an AFS client.

4.3 Meeting AFS Goals

The AFS WADFS design, as described in this chapter, serves to meet the system goals stated in Chapter 3. This section revisits each of these AFS goals, and identifies the specific architectural constructs that bear on them.

4.3.1 Scale

To date, AFS has been deployed to over 140 sites world-wide, with approximately 60 of these cells visible on the public Internet. AFS sites are currently operating in several European countries, in Japan, and in Australia. While many sites are modest in size, certain cells contain more than 30,000 accounts. AFS sites have realized client/server ratios in excess of the targeted 200:1.

4.3.2 Name Space

A single uniform name space has been constructed across all cells in the greater AFS user community. Any pathname beginning with */afs* may indeed be used at any AFS client. A set of common conventions regarding the organization of the top-level */afs* directory and several directories below it have been established. These conventions also assist in the location of certain per-cell resources, such as AFS configuration files.

Both access transparency and location transparency are supported by AFS, as evidenced by the common access mechanisms and by the ability to transparently relocate volumes.

4.3.3 Performance

AFS employs caching extensively at all levels to reduce the cost of "remote" references. Measured data cache hit ratios are very high, often over 95%. This indicates that the file images kept on local disk are very effective in satisfying the set of remote file references generated by clients. The introduction of file system callbacks has also been demonstrated to be very effective in the efficient implementation of cache synchronization. Replicating files and system databases across multiple server machines distributes load among the given servers. The *Rx* RPC subsystem has operated successfully at network speeds ranging from 19.2 kilobytes/second to experimental gigabit/second FDDI networks.

Even at the intra-site level, AFS has been shown to deliver good performance, especially in high-load situations. One often-quoted study [1] compared the performance of an older version of AFS with that of NFS on a large file system task named the **Andrew Benchmark**. While NFS sometimes outperformed AFS at low load levels, its performance fell off rapidly at higher loads while AFS performance degradation was not significantly affected.

4.3.4 Security

The use of Kerberos as the AFS authentication system fits the security goal nicely. Access to AFS files from untrusted client machines is predicated on the caller's possession of the appropriate Kerberos ticket(s). Setting up per-site, Kerveros-based authentication services compartmentalizes any security breach to the cell which was compromised. Since the *Cache Manager* will store multiple tickets for its users, they may take on different identities depending on the set of file servers being accessed.

4.3.5 Access Control

AFS extends the standard UNIX authorization mechanism with per-directory Access Control Lists. These ACLs allow specific AFS principals and groups of these principals to be granted a wide variety of rights on the associated files. Users may create and manipulate AFS group entities without administrative assistance, and place these tailored groups on ACLs.

4.3.6 Reliability

A subset of file server crashes are masked by the use of read-only replication on volumes containing slowly-changing files. Availability of important, frequently-used programs such as editors and compilers may thus been greatly improved. Since the level of replication may be chosen per volume, and easily changed, each site may decide the proper replication levels for certain programs and/or data.

Similarly, replicated system databases help to maintain service in the face of server crashes and network partitions.

4.3.7 Administrability

Such features as pervasive, secure RPC interfaces to all AFS system components, volumes, overseer processes for monitoring and management of file system agents, intelligent user-level access tools, interface routines providing performance and statistics information, and an automated backup service tailored to a volume-based environment all contribute to the administrability of the AFS system.

4.3.8 Interoperability/Coexistence

Due to its VFS-style implementation, the AFS client code may be easily installed in the machine's kernel, and may service file requests without interfering in the operation of any other installed file system. Machines either not capable of running AFS natively or choosing not to do so may still access AFS files via NFS with the help of a protocol translator agent.

4.3.9 Heterogeneity/Portability

As most modern kernels use a VFS-style interface to support their native file systems, AFS may usually be ported to a new hardware and/or software environment in a relatively straightforward fashion. Such ease of porting allows AFS to run on a wide variety of platforms.

Chapter 5

Future AFS Design Refinements

5.1 Overview

The current AFS WADFS design and implementation provides a high-performance, scalable, secure, and flexible computing environment. However, there is room for improvement on a variety of fronts. This chapter considers a set of topics, examining the shortcomings of the current AFS system and considering how additional functionality may be fruitfully constructed.

Many of these areas are already being addressed in the next-generation AFS system which is being built as part of Open Software Foundation's (OSF) Distributed Computing Environment [7] [8].

5.2 UNIX Semantics

Any distributed file system which extends the UNIX file system model to include remote file accesses presents its application programs with failure modes which do not exist in a single-machine UNIX implementation. This semantic difference is difficult to mask.

The current AFS design varies from pure UNIX semantics in other ways. In a single-machine UNIX environment, modifications made to an open file are immediately visible to other processes with open file descriptors to the same file. AFS does not reproduce this behavior when programs on different machines access the same file. Changes made to one cached copy of the file are not made immediately visible to other cached copies. The changes are only made visible to other access sites when a modified version of a file is stored back to the server providing its primary disk storage. Thus, one client's changes may be entirely overwritten by another client's modifications. The situation is further complicated by the possibility that dirty file chunks may be flushed out to the *File Server* before the file is closed.

The version of AFS created for the OSF offering extends the current, untyped callback notion to a set of multiple, independent synchronization guarantees. These synchronization *tokens* allow functionality not offered by AFS-3, including byte-range mandatory locking, exclusive file opens, and read and write privileges over portions of a file.

5.3 Improved Name Space Management

Discovery of new AFS cells and their integration into each existing cell's name space is a completely manual operation in the current system. As the rate of new cell creations increases, the load imposed on system administrators also increases. Also, representing each cell's file space entry as a mount point object in the */afs* directory leads to a potential problem. As the number of entries in the */afs* directory increase, search time through the directory also grows.

One improvement to this situation is to implement the top-level */afs* directory through a Domain-style database. The database would map cell names to the set of server machines providing authentication and volume location services for that cell. The *Cache Manager* would query the cell database in the course of pathname resolution, and cache its lookup results.

In this database-style environment, adding a new cell entry under */afs* is accomplished by creating the appropriate database entry. The new cell information is then immediately accessible to all AFS clients.

5.4 Read/Write Replication

The AFS-3 servers and databases are currently equipped to handle read/only replication exclusively. However, other distributed file systems have demonstrated the feasibility of providing full read/write replication of data in environments very similar to AFS [11]. Such systems can serve as models for the set of required changes.

5.5 Disconnected Operation

Several facilities are provided by AFS so that server failures and network partitions may be completely or partially masked. However, AFS does *not* provide for completely **disconnected operation** of file system clients. Disconnected operation is a mode in which a client continues to access critical data during accidental or intentional inability to access the shared file repository. After some period of autonomous operation on the set of cached files, the client reconnects with the repository and resynchronizes the contents of its cache with the shared store.

Studies of related systems provide evidence that such disconnected operation is feasible [11] [12]. Such a capability may be explored for AFS.

5.6 Multiprocessor Support

The LWP lightweight thread package used by all AFS system processes assumes that individual threads may execute non-preemeptively, and that all other threads are quiescent until control is explicitly relinquished from within the currently active thread. These assumptions conspire to prevent AFS from operating correctly on a multiprocessor platform.

A solution to this restriction is to restructure the AFS code organization so that the proper locking is performed. Thus, critical sections which were previously only implicitly defined are explicitly specified.

Bibliography

[1] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West, *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 51-81.

[2] Michael L. Kazar, *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[3] Alfred Z. Spector, Michael L. Kazar, *Uniting File Systems*, Unix Review, March 1989,

[4] Johna Till Johnson, *Distributed File System Brings LAN Technology to WANs*, Data Communications, November 1990, pp. 66-67.

[5] Michael Padovano, PADCOM Associates, *AFS widens your horizons in distributed computing*, Systems Integration, March 1991.

[6] Steve Lammert, *The AFS 3.0 Backup System*, LISA IV Conference Proceedings, Colorado Springs, Colorado, October 1990.

[7] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, Edward R. Zayas, *DEcorum File System Architectural Overview*, USENIX Conference Proceedings, Anaheim, Texas, Summer 1990.

[8] *AFS Drives DCE Selection*, Digital Desktop, Vol. 1, No. 6, September 1990.

[9] Levine, P.H., *The Apollo DOMAIN Distributed File System*, in *NATO ASI Series: Theory and Practice of Distributed Operating Systems*, Y. Paker, J-P. Banatre, M. Bozyigit, editors, Springer-Verlag, 1987.

[10] M.N. Nelson, B.B. Welch, J.K. Ousterhout, *Caching in the Sprite Network File System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988.

[11] James J. Kistler, M. Satyanarayanan, *Disconnected Operaton in the Coda File System*, CMU School of Computer Science technical report, CMU-CS-91-166, 26 July 1991.

[12] Puneet Kumar, M. Satyanarayanan, *Log-Based Directory Resolution in the Coda File System*, CMU School of Computer Science internal document, 2 July 1991.

[13] Sun Microsystems, Inc., *NFS: Network File System Protocol Specification*, RFC 1094, March 1989.

[14] Sun Microsystems, Inc,. *Design and Implementation of the Sun Network File System*, USENIX Summer Conference Proceedings, June 1985.

[15] C.H. Sauer, D.W Johnson, L.K. Loucks, A.A. Shaheen-Gouda, and T.A. Smith, *RT PC Distributed Services Overview*, Operating Systems Review, Vol. 21, No. 3, July 1987.

[16] A.P. Rifkin, M.P. Forbes, R.L. Hamilton, M. Sabrio, S. Shah, and K. Yueh, *RFS Architectural Overview*, Usenix Conference Proceedings, Atlanta, Summer 1986.

[17] Edward R. Zayas, *Administrative Cells: Proposal for Cooperative Andrew File Systems*, Information Technology Center internal document, Carnegie Mellon University, 25 June 1987.

[18] Ed. Zayas, Craig Everhart, *Design and Specification of the Cellular Andrew Environment*, Information Technology Center, Carnegie Mellon University, CMU-ITC-070, 2 August 1988.

[19] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Ubik - A Library For Managing Ubiquitous Data*, ITCID, Pittsburgh, PA, Month, 1988.

[20] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Quorum Completion*, ITCID, Pittsburgh, PA, Month, 1988.

[21] S. R. Kleinman. *Vnodes: An Architecture for Multiple file System Types in Sun UNIX*, Conference Proceedings, 1986 Summer Usenix Technical Conference, pp. 238-247, El Toro, CA, 1986.

[22] S.P. Miller, B.C. Neuman, J.I. Schiller, J.H. Saltzer. *Kerberos Authentication and Authorization System*, Project Athena Technical Plan, Section E.2.1, M.I.T., December 1987.

[23] Bill Bryant. *Designing an Authentication System: a Dialogue in Four Scenes*, Project Athena internal document, M.I.T, draft of 8 February 1988.

Specification for the Rx RPC Facility

Chapter 1

Overview

1.1 Introduction

The *Rx* package provides a high-performance, multi-threaded, and secure mechanism by which remote procedure calls (RPCs) may be performed between programs executing anywhere in a network of computers. The *Rx* protocol is adaptive, conforming itself to widely varying network communication media. It allows user applications to define and insert their own security modules, allowing them to execute the precise end-to-end authentication algorithms required to suit their needs and goals. Although pervasive throughout the AFS distributed file system, all of its agents, and many of its standard application programs, *Rx* is entirely separable from AFS and does not depend on any of its features. In fact, *Rx* can be used to build applications engaging in RPC-style communication under a variety of UNIX-style file systems. There are in-kernel and user-space implementations of the *Rx* facility, with both sharing the same interface.

This document provides a comprehensive and detailed treatment of the *Rx* RPC package.

1.2 Basic Concepts

The *Rx* design operates on the set of basic concepts described in this section.

1.2.1 Security

The *Rx* architecture provides for tight integration between the RPC mechanism and methods for making this communication medium secure. As elaborated in Section 5.3.1.3 and illustrated by the built-in *rxkad* security system described in Chapter 3, *Rx* defines the format for a generic security module, and then allows application programmers to define and activate instantiations of these modules. *Rx* itself knows nothing about the internal details of any particular security model, or the module-specific state it requires. It does, however, know when to call the generic security operations, and so can easily execute the security algorithm defined. *Rx* does maintain basic state per connection on behalf of any given security class.

1.2.2 Services

An *Rx*-based server exports *services*, or specific RPC interfaces that accomplish certain tasks. Services are identified by *(host-address, UDP-port, serviceID)* triples. An *Rx* service is installed and initialized on a given host through the use of the *rx_NewService()* routine (See Section 5.6.3). Incoming calls are stamped with the *Rx* service type, and must match an installed service to be accepted. Internally, *Rx* services also carry string names which identify them, which is useful for remote debugging and statistics-gathering programs. The use of a service ID allows a single server process to export multiple, independently-specified *Rx* RPC services.

Each *Rx* service contains one or more *security classes*, as implemented by individual *security objects*. These security objects implement end-to-end security protocols. Individual peer-to-peer *connections* established on behalf of an *Rx* service will select exactly one of the supported security objects to define the authentication procedures followed by all calls associated with the connection. Applications are not limited to using only the core set of built-in security objects offered by *Rx*. They are free to define their own security objects in order to execute the specific protocols they require.

It is possible to specify both the minimum and maximum number of lightweight processes available to handle simultaneous calls directed to an *Rx* service. In addition, certain procedures may be registered with the service and called at specific times in the course of handling an RPC request.

1.2.3 Connections

An *Rx* connection represents an authenticated communication path, allowing a sequence of multiple asynchronous conversations (*calls*). Each connection is identified by a connection ID. The low-order bits of the connection ID are reserved so that they may be stamped with the index of a particular call channel. With up to `RX_MAXCALLS` concurrent calls (set to 4 in this implementation), the bottom two bits are set aside for this purpose. The connection ID is not sufficient to uniquely identify an *Rx* connection by itself. Should a client crash and restart, it may reuse a connection ID, causing inconsistent results. Included with the connection ID is the **epoch**, or start time for the client side of the connection. After a crash, the next incarnation of the client will choose a different epoch value. This will differentiate the new incarnation from the orphaned connection record on the server side.

Each connection is associated with a parent service, which defines a set of supported security models. At creation time, an *Rx* connection selects the particular security protocol it will implement, referencing the associated service. The connection structure maintains state for each individual call simultaneously handled.

1.2.4 Peers

For each connection, *Rx* maintains information describing the entity, or **peer**, on the other side of the wire. A peer is identified by a *(host, UDP-port)* pair, with an IP address used to identify the *host*. Included in the information kept on this remote communication endpoint are such network parameters as the maximum packet size supported by the host, current readings on round trip time and retransmission delays, and **packet skew** (see Section 1.2.7). There are also congestion control fields, including retransmission statistics and descriptions of the maximum number of packets that may be sent to the peer without pausing. Peer structures are shared between connections whenever possible, and, hence, are reference-counted. A peer object may be garbage-collected if it is not actively referenced by any connection structure and a sufficient period of time has lapsed since the reference count dropped to zero.

1.2.5 Calls

An *Rx* call represents an individual RPC being executed on a given connection. As described above, each connection may have up to `RX_MAXCALLS` calls active at any one instant. The information contained in each call structure is specific to the given call.

"Permanent" call state, such as the call number, is maintained in the connection structure itself.

1.2.6 Quotas

Each attached server thread must be able to make progress to avoid system deadlock. The *Rx* facility ensures that it can always handle the arrival of the next unacknowledged data packet for an attached call with its system of **packet quotas**. A certain number of packets are reserved per server thread for this purpose, allowing the server threads to queue up an entire window full of data for an active call and still have packet buffers left over to be able to read its input without blocking.

1.2.7 Packet Skew

If a packet is received *n* packets later than expected (based on packet serial numbers), then we define it to have a skew of *n*. The maximum skew values allow us to decide when a packet hasn't been received yet because it is out of order, as opposed to when it is likely to have been dropped.

1.2.8 Multicasting

The *rx_multi.c* module provides for multicast abilities, sending an RPC to several targets simultaneously. While true multicasting is not achieved, it is simulated by a rapid succession of packet transmissions and a collection algorithm for the replies. A client program, though, may be programmed as if multicasting were truly taking place. Thus, *Rx* is poised to take full advantage of a system supporting true multicasting with minimal disruption to the existing client code base.

1.3 Scope

This paper is a member of a documentation suite providing specifications as to the operation and interfaces offered by the various AFS servers and agents. *Rx* is an integral part of the AFS environment, as it provides the high-performance, secure pathway by which these system components communicate across the network. Although AFS is dependent on *Rx*'s services, the reverse is not true. *Rx* is a fully independent RPC package, standing on its own and usable in other environments.

The intent of this work is to provide readers with a sufficiently detailed description of *Rx* that they may proceed to write their own applications on top of it. In fact, code for a sample *Rx* server and client are provided.

One topic related to *Rx* will *not* be covered by this document, namely the *Rxgen* stub generator. Rather, *rxgen* is addressed in a separate document.

1.4 Document Layout

After this introduction, Chapter 2 will introduce and describe various facilities and tools that support *Rx*. In particular, the threading and locking packages used by *Rx* will be examined, along with a set of timer and preemption tools. Chapter 3 proceeds to examine the details of one of the built-in security modules offered by *Rx*. Based on the Kerberos system developed by MIT's Project Athena, this *rxkad* module allows secure, ecrypted communication between the server and client ends of the RPC. Chapter 5 then provides the full *Rx* programming interface, and Chapter 6 illustrates the use of this programming interface by providing a fully-operational programming example employing *Rx*. This *rxdemo* suite is examined in detail, ranging all the way from a step-by-step analysis of the human-authored files, and the *Rxgen*-generated files upon which they are based, to the workings of the associated *Makefile*. Output from the example *rxdemo* server and client is also provided.

1.5 Related Documents

Titles for the full suite of AFS specification documents are listed below. All of the servers and agents making up the AFS computing environment, whether running in the UNIX kernel or in user space, utilize an *Rx* RPC interface through which they export their services.

- *AFS-3 Programmer's Reference: Architectural Overview*: This paper provides an architectual overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and examining their interactions.

- *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*: This document describes the workings and interfaces of the two primary AFS agents, the *File Server* and *Cache Manager*. The *File Server* provides a centralized disk repository for sets of files, regulating access to them. End users sitting on client machines rely on the *Cache Manager* agent, running in their kernel, to act as their agent in accessing the data stored on *File Server* machines, making those files appear as if they were really housed locally.
- *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.
- *AFS-3 Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for mapping printable user names to and from their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users, which are suitable for placement on access control lists (ACLs).
- *AFS-3 Programmer's Reference: BOS Server Interface*: This paper explicates the "nanny" service which assists in the administrability of the AFS environment.

In addition to these papers, the AFS 3.1 product is delivered with its own user, system administrator, installation, and command reference documents.

Chapter 2

The LWP Lightweight Process Package

2.1 Introduction

This chapter describes a package allowing multiple threads of control to coexist and cooperate within one UNIX process. Each such thread of control is also referred to as a *lightweight process*, in contrast to the traditional UNIX (heavyweight) process. Except for the limitations of a fixed stack size and non-preemptive scheduling, these lightweight processes possess all the properties usually associated with full-fledged processes in typical operating systems. For the purposes of this document, the terms **lightweight process**, **LWP**, and **thread** are completely interchangeable, and they appear intermixed in this chapter. Included in this lightweight process facility are various sub-packages, including services for locking, I/O control, timers, fast time determination, and preemption.

The *Rx* facility is not the only client of the *LWP* package. Other LWP clients within AFS include the *File Server*, *Protection Server*, *BOS Server*, *Volume Server*, *Volume Location Server*, and the *Authentication Server*, along with many of the AFS application programs.

2.2 Description

2.2.1 LWP Overview

The *LWP* package implements primitive functions that provide the basic facilities required to enable procedures written in C to execute concurrently and asynchronously. The *LWP* package is meant to be general-purpose (note the applications mentioned above), with a heavy emphasis on simplicity. Interprocess communication facilities can be built on top of this basic mechanism and in fact, many different IPC mechanisms could be implemented.

In order to set up the threading support environment, a one-time invocation of the *LWP_InitializeProcessSupport()* function must precede the use of the facilities described here. This initialization function carves an initial process out of the currently executing C procedure and returns its thread ID. For symmetry, an *LWP_TerminateProcessSupport()* function may be used explicitly to release any storage allocated by its counterpart. If this function is used, it must be issued from the thread created by the original *LWP_InitializeProcessSupport()* invocation.

When any of the lightweight process functions completes, an integer value is returned to indicate whether an error condition was encountered. By convention, a return value of zero indicates that the operation succeeded.

Macros, typedefs, and manifest constants for error codes needed by the threading mechanism are exported by the *lwp.h* include file. A lightweight process is identified by an object of type `PROCESS`, which is defined in the include file.

The process model supported by the *LWP* operations is based on a *non-preemptive priority dispatching* scheme. A *priority* is an integer in the range [`0..LWP_MAX_PRIORITY`], where 0 is the lowest priority. Once a given thread is selected and dispatched, it remains in control until it voluntarily relinquishes its claim on the CPU. Control may be relinquished by either explicit means (*LWP_DispatchProcess()*) or implicit means (through the use of certain other *LWP* operations with this side effect). In general, all *LWP* operations that may cause a higher-priority process to become ready for dispatching preempt the process requesting the service. When this occurs, the dispatcher mechanism takes over and automatically schedules the highest-priority runnable process. Routines in this category, where the scheduler is guaranteed to be invoked in the absence of errors, are:

- *LWP_WaitProcess()*
- *LWP_MwaitProcess()*

- *LWP_SignalProcess()*
- *LWP_DispatchProcess()*
- *LWP_DestroyProcess()*

The following functions are guaranteed **not** to cause preemption, and so may be issued with no fear of losing control to another thread:

- *LWP_InitializeProcessSupport()*
- *LWP_NoYieldSignal()*
- *LWP_CurrentProcess()*
- *LWP_ActiveProcess()*
- *LWP_StackUsed()*
- *LWP_NewRock()*
- *LWP_GetRock()*

The symbol `LWP_NORMAL_PRIORITY`, whose value is `(LWP_MAX_PRIORITY-2)`, provides a reasonable default value to use for process priorities.

The `lwp_debug` global variable can be set to activate or deactivate debugging messages tracing the flow of control within the *LWP* routines. To activate debugging messages, set *lwp_debug* to a non-zero value. To deactivate, reset it to zero. All debugging output from the LWP routines is sent to `stdout`.

The *LWP* package checks for stack overflows at each context switch. The variable that controls the action of the package when an overflow occurs is *lwp_overflowAction*. If it is set to `LWP_SOMESSAGE`, then a message will be printed on `stderr` announcing the overflow. If `lwp_overflowAction` is set to `LWP_SOABORT`, the *abort*() *LWP* routine will be called. Finally, if `lwp_overflowAction` is set to `LWP_SOQUIET`, the *LWP* facility will ignore the errors. By default, the `LWP_SOABORT` setting is used.

Here is a sketch of a simple program (using some psuedocode) demonstrating the high-level use of the *LWP* facility. The opening `#include` line brings in the exported *LWP* definitions. Following this, a routine is defined to wait on a “queue” object until something is deposited in it, calling the scheduler as soon as something arrives. Please note that various *LWP* routines are introduced here. Their definitions will appear later, in Section 2.3.1.

```
#include <afs/lwp.h>

static read_process(id)
    int *id;

{
    /*
     * Just relinquish control for now
     */
    LWP_DispatchProcess();

    for (;;) {
        /*
         * Wait until there is something in the queue
         */
        while (empty(q))
            LWP_WaitProcess(q);

        /*
         * Process the newly-arrived queue entry
         */
        LWP_DispatchProcess();
    }
}
```

The next routine, *write_process()*, sits in a loop, putting messages on the shared queue and signalling the reader, which is waiting for activity on the queue. Signalling a thread is accomplished via the *LWP_SignalProcess()* library routine.

```
static write_process()

{
    . . .

    /*
     * Loop, writing data to the shared queue.
     */
    for (mesg = messages; *mesg != 0; mesg++) {
        insert(q, *mesg);
        LWP_SignalProcess(q);
    }
}
```

Finally, here is the main routine for this demo pseudocode. It starts by calling the *LWP* initialization routine. Next, it creates some number of reader threads with calls to *LWP_CreateProcess()* in addition to the single writer thread. When all threads terminate, they will signal the main routine on the `done` variable. Once signalled, the main routine will reap all the threads with the help of the *LWP_DestroyProcess()* function.

```
main(argc, argv)
    int argc;
    char **argv;

{
    PROCESS *id;        /*Initial thread ID*/

    /*
     * Set up the LWP package, create the initial thread ID.
     */
    LWP_InitializeProcessSupport(0, &id);

    /*
     * Create a set of reader threads.
     */
    for (i = 0; i < nreaders; i++)
        LWP_CreateProcess(read_process,
                          STACK_SIZE,
                          0,
                          i,
                          "Reader",
                          &readers[i]);

    /*
     * Create a single writer thread.
     */
    LWP_CreateProcess(write_process,
                      STACK_SIZE,
                      1,
                      0,
                      "Writer",
                      &writer);

    /*
     * Wait for all the above threads to terminate.
     */
    for (i = 0; i <= nreaders; i++)
        LWP_WaitProcess(&done);

    /*
     * All threads are done.  Destroy them all.
     */
    for (i = nreaders-1; i >= 0; i--)
        LWP_DestroyProcess(readers[i]);
}
```

2.2.2 Locking

The *LWP* locking facility exports a number of routines and macros that allow a C programmer using *LWP* threading to place read and write locks on shared data structures. This locking facility was also written with simplicity in mind.

In order to invoke the locking mechanism, an object of type `struct Lock` must be associated with the object. After being initialized with a call to `LockInit()`, the lock object is used in invocations of various macros, including *ObtainReadLock()*, *ObtainWriteLock()*, *ReleaseReadLock()*, *ReleaseWriteLock()*, *ObtainSharedLock()*, *ReleaseSharedLock()*, and *BoostSharedLock()*.

Lock semantics specify that any number of readers may hold a lock in the absence of a writer. Only a single writer may acquire a lock at any given time. The lock package guarantees fairness, legislating that each reader and writer will *eventually* obtain a given lock. However, this fairness is only guaranteed if the priorities of the competing processes are identical. Note that ordering is *not* guaranteed by this package.

Shared locks are read locks that can be "boosted" into write locks. These shared locks have an unusual locking matrix. Unboosted shared locks are compatible with read locks, yet incompatible with write locks and other shared locks. In essence, a thread holding a shared lock on an object has effectively read-locked it, and has the option to promote it to a write lock without allowing any other writer to enter the critical region during the boost operation itself.

It is illegal for a process to request a particular lock more than once without first releasing it. Failure to obey this restriction will cause deadlock. This restriction is not enforced by the LWP code.

Here is a simple pseudocode fragment serving as an example of the available locking operations. It defines a `struct Vnode` object, which contains a lock object. The *get_vnode()* routine will look up a `struct Vnode` object by name, and then either read-lock or write-lock it.

As with the high-level LWP example above, the locking routines introduced here will be fully defined later, in Section 2.3.2.

```
#include <afs/lock.h>

struct Vnode {
    . . .
    struct Lock lock;   /* Used to lock this vnode */
    . . .
};

#define READ    0
#define WRITE   1

struct Vnode *get_vnode(name, how)
    char *name;
```

```
    int how;
{
    struct Vnode *v;

    v = lookup(name);
    if (how == READ)
        ObtainReadLock(&v->lock);
    else
        ObtainWriteLock(&v->lock);
}
```

2.2.3 IOMGR

The *IOMGR* facility associated with the *LWP* service allows threads to wait on various UNIX events. The exported *IOMGR_Select()* routine allows a thread to wait on the same set of events as the UNIX *select()* call. The parameters to these two routines are identical. *IOMGR_Select()* puts the calling LWP to sleep until no threads are active. At this point, the built-in IOMGR thread, which runs at the lowest priority, wakes up and coalesces all of the select requests together. It then performs a single *select()* and wakes up all threads affected by the result.

The *IOMGR_Signal()* routine allows an LWP to wait on the delivery of a UNIX signal. The IOMGR thread installs a signal handler to catch all deliveries of the UNIX signal. This signal handler posts information about the signal delivery to a global data structure. The next time that the IOMGR thread runs, it delivers the signal to any waiting LWP.

Here is a pseudocode example of the use of the *IOMGR* facility, providing the blueprint for an implemention a thread-level socket listener.

```
void rpc_SocketListener()
{
    int ReadfdMask, WritefdMask, ExceptfdMask, rc;
    struct timeval *tvp;

    while(TRUE) {
        . . .
        ExceptfdMask = ReadfdMask = (1 << rpc_RequestSocket);
        WritefdMask = 0;

        rc = IOMGR_Select(8*sizeof(int),
                          &ReadfdMask,
                          &WritefdMask,
                          &ExceptfdMask,
```

```
                        tvp);

        switch(rc) {
            case 0:     /*Timeout*/
                continue;       /*Main while loop*/

            case -1:    /*Error*/
                SystemError("IOMGR_Select");
                exit(-1);

            case 1:     /*RPC packet arrived!*/
                . . . process packet . . .
                break;

            default:    /*Should never occur*/
        }

    }

}
```

2.2.4 Timer

The timer package exports a number of routines that assist in manipulating lists of objects of type `struct TM_Elem`. These `struct TM_Elem` timers are assigned a timeout value by the user and inserted in a package-maintained list. The time remaining to each timer's timeout is kept up to date by the package under user control. There are routines to remove a timer from its list, to return an expired timer from a list, and to return the next timer to expire.

A timer is commonly used by inserting a field of type `struct TM_Elem` into a structure. After setting the desired timeout value, the structure is inserted into a list by means of its timer field.

Here is a simple pseudocode example of how the timer package may be used. After calling the package initialization function, *TM_Init()*, the pseudocode spins in a loop. First, it updates all the timers via *TM_Rescan()* calls. Then, it pulls out the first expired timer object with *TM_GetExpired()* (if any), and processes it.

```
static struct TM_Elem *requests;

. . .

TM_Init(&requests);        /*Initialize timer list*/
. . .
```

```
for (;;) {
    TM_Rescan(requests);   /* Update the timers */
    expired = TM_GetExpired(requests);
    if (expired == 0)
        break;
    . . . process expired element . . .
    }
```

2.2.5 Fast Time

The fast time routines allows a caller to determine the current time of day without incurring the expense of a kernel call. It works by mapping the page of the kernel that holds the time-of-day variable and examining it directly. Currently, this package only works on Suns. The routines may be called on other architectures, but they will run more slowly.

The initialization routine for this package is fairly expensive, since it does a lookup of a kernel symbol via *nlist()*. If the client application program only runs for only a short time, it may wish to call *FT_Init()* with the `notReally` parameter set to TRUE in order to prevent the lookup from taking place. This is useful if you are using another package that uses the fast time facility.

2.2.6 Preemption

The preemption package provides a mechanism by which control can pass between lightweight processes without the need for explicit calls to *LWP_DispatchProcess()*. This effect is achieved by periodically interrupting the normal flow of control to check if other (higher priority) procesess are ready to run.

The package makes use of the BSD interval timer facilities, and so will cause programs that make their own use of these facilities to malfunction. In particular, use of *alarm(3)* or explicit handling of `SIGALRM` is disallowed. Also, calls to *sleep(3)* may return prematurely.

Care should be taken that routines are re-entrant where necessary. In particular, note that *stdio(3)* is not re-entrant in general, and hence multiple threads performing I/O on the same `FILE` structure may function incorrectly.

An example pseudocode routine illustrating the use of this preemption facility appears below.

```
#include <sys/time.h>
#include "preempt.h"

        ...
        struct timeval tv;

        LWP_InitializeProcessSupport( ... );
        tv.tv_sec = 10;
        tv.tv_usec = 0;
        PRE_InitPreempt(&tv);
        PRE_PreemptMe();
        ...
        PRE_BeginCritical();
        ...
        PRE_EndCritical();
        ...
        PRE_EndPreempt();
```

2.3 Interface Specifications

2.3.1 LWP

This section covers the calling interfaces to the *LWP* package. Please note that *LWP* macros (e.g., *ActiveProcess*) are also included here, rather than being relegated to a different section.

2.3.1.1 LWP_InitializeProcessSupport — Initialize the *LWP* package

```
int LWP_InitializeProcessSupport(IN int priority;
                                 OUT PROCESS *pid)
```

Description

This function initializes the *LWP* package. In addition, it turns the current thread of control into the initial process with the specified `priority`. The process ID of this initial thread is returned in the `pid` parameter. This routine must be called before any other routine in the *LWP* library. The scheduler will NOT be invoked as a result of calling *LWP_InitializeProcessSupport()*.

Error Codes

`LWP_EBADPRI` The given `priority` is invalid, either negative or too large.

2.3.1.2 LWP_TerminateProcessSupport — End process support, perform cleanup

```
int LWP_TerminateProcessSupport()
```

Description

This routine terminates the *LWP* threading support and cleans up after it by freeing any auxiliary storage used. This routine must be called from within the process that invoked *LWP_InitializeProcessSupport()*. After *LWP_TerminateProcessSupport()* has been called, it is acceptable to call *LWP_InitializeProcessSupport()* again in order to restart *LWP* process support.

Error Codes

`---` Always succeeds, or performs an *abort()*.

2.3.1.3 LWP_CreateProcess — Create a new thread

```
int LWP_CreateProcess(IN int (*ep)();
                      IN int stacksize;
                      IN int priority;
```

```
IN char *parm;
IN char *name;
OUT PROCESS *pid)
```

Description

This function is used to create a new lightweight process with a given printable `name`. The `ep` argument identifies the function to be used as the body of the thread. The argument to be passed to this function is contained in `parm`. The new thread's stack size in bytes is specified in `stacksize`, and its execution priority in `priority`. The `pid` parameter is used to return the process ID of the new thread.

If the thread is successfully created, it will be marked as runnable. The scheduler is called before the *LWP_CreateProcess()* call completes, so the new thread may indeed begin its execution before the completion. Note that the new thread is guaranteed *NOT* to run before the call completes if the specified `priority` is lower than the caller's. On the other hand, if the new thread's priority is higher than the caller's, then it is guaranteed to run before the creation call completes.

Error Codes

`LWP_EBADPRI` The given `priority` is invalid, either negative or too large.

`LWP_NOMEM` Could not allocate memory to satisfy the creation request.

2.3.1.4 LWP_DestroyProcess — Create a new thread

`int` **LWP_DestroyProcess**`(IN PROCESS pid)`

Description

This routine destroys the thread identified by `pid`. It will be terminated immediately, and its internal storage will be reclaimed. A thread is allowed to destroy itself. In this case, of course, it will only get to see the return code if the operation fails. Note that a thread may also destroy itself by returning from the parent C routine.

The scheduler is called by this operation, which may cause an arbitrary number of threads to execute before the caller regains the processor.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

2.3.1.5 LWP_WaitProcess — Wait on an event

```
int LWP_WaitProcess(IN char *event)
```

Description

This routine puts the thread making the call to sleep until another LWP calls the *LWP_SignalProcess()* or *LWP_NoYieldSignal()* routine with the specified event. Note that signalled events are not queued. If a signal occurs and no thread is awakened, the signal is lost. The scheduler is invoked by the *LWP_WaitProcess()* routine.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

`LWP_EBADEVENT` The given `event` pointer is null.

2.3.1.6 LWP_MwaitProcess — Wait on a set of events

```
int LWP_MwaitProcess(IN int wcount;
                     IN char *evlist[])
```

Description

This function allows a thread to wait for `wcount` signals on any of the items in the given `evlist`. Any number of signals of a particular event are only counted once. The `evlist` is a null-terminated list of events to wait for. The scheduler will be invoked.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

`LWP_EBADCOUNT` An illegal number of events has been supplied.

2.3.1.7 LWP_SignalProcess — Signal an event

`int` **LWP_SignalProcess**`(IN char *event)`

Description

This routine causes the given `event` to be signalled. All threads waiting for this event (exclusively) will be marked as runnable, and the scheduler will be invoked. Note that threads waiting on *multiple* events via *LWP_MwaitProcess()* may *not* be marked as runnable. Signals are not queued. Therefore, if no thread is waiting for the signalled event, the signal will be lost.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

`LWP_EBADEVENT` A null `event` pointer has been provided.

`LWP_ENOWAIT` No thread was waiting on the given `event`.

2.3.1.8 LWP_NoYieldSignal — Signal an event without invoking scheduler

`int` **LWP_NoYieldSignal**`(IN char *event)`

Description

This function is identical to *LWP_SignalProcess()* except that the scheduler will not be invoked. Thus, control will remain with the signalling process.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

`LWP_EBADEVENT` A null `event` pointer has been provided.

`LWP_ENOWAIT` No thread was waiting on the given `event`.

2.3.1.9 LWP_DispatchProcess — Yield control to the scheduler

`int` **LWP_DispatchProcess**`()`

Description

This routine causes the calling thread to yield voluntarily to the LWP scheduler. If no other thread of appropriate priority is marked as runnable, the caller will continue its execution.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

2.3.1.10 LWP_CurrentProcess — Get the current thread's ID

`int` **LWP_CurrentProcess**`(IN PROCESS *pid)`

Description

This call places the current lightweight process ID in the `pid` parameter.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

2.3.1.11 LWP_ActiveProcess — Get the current thread's ID (macro)

`int` **LWP_ActiveProcess**`()`

Description

This macro's value is the current lightweight process ID. It generates a value identical to that acquired by calling the *LWP_CurrentProcess()* function described above if the *LWP* package has been initialized. If no such initialization has been done, it will return a value of zero.

2.3.1.12 LWP_StackUsed — Calculate stack usage

```
int LWP_StackUsed(IN PROCESS pid;
                  OUT int *max;
                  OUT int *used)
```

Description

This function returns the amount of stack space allocated to the thread whose identifier is `pid`, and the amount actually used so far. This is possible if the global variable `lwp_stackUseEnabled` was TRUE when the thread was created (it is set this way by default). If so, the thread's stack area was initialized with a special pattern. The memory still stamped with this pattern can be determined, and thus the amount of stack used can be calculated. The `max` parameter is always set to the thread's stack allocation value, and `used` is set to the computed stack usage if `lwp_stackUseEnabled` was set when the process was created, or else zero.

Error Codes

`LWP_NO_STACK` Stack usage was not enabled at thread creation time.

2.3.1.13 LWP_NewRock — Establish thread-specific storage

```
int LWP_NewRock(IN int tag;
                IN char **value)
```

Description

This function establishes a "rock", or thread-specific information, associating it with the calling LWP. The `tag` is intended to be any unique integer value, and the `value` is a pointer to a character array containing the given data.

Users of the *LWP* package must coordinate their choice of `tag` values. Note that a `tag`'s value cannot be changed. Thus, to obtain a mutable data structure, another level of indirection is required. Up to `MAXROCKS` (4) rocks may be associated with any given thread.

Error Codes

LWP_EBADROCK A rock with the given `tag` field already exists.

LWP_ENOROCKS All of the `MAXROCKS` are in use.

2.3.1.14 LWP_GetRock — Retrieve thread-specific storage

```
int LWP_GetRock(IN int tag;
                OUT **value)
```

Description

This routine recovers the thread-specific information associated with the calling process and the given `tag`, if any. Such a rock had to be established through a *LWP_NewRock()* call. The rock's value is deposited into `value`.

Error Codes

LWP_EBADROCK A rock has not been associated with the given `tag` for this thread.

2.3.2 Locking

This section covers the calling interfaces to the locking package. Many of the user-callable routines are actually implemented as macros.

2.3.2.1 Lock_Init — Initialize lock structure

```
void Lock_Init(IN struct Lock *lock)
```

Description

This function must be called on the given `lock` object before any other operations can be performed on it.

Error Codes

--- No value is returned.

2.3.2.2 ObtainReadLock — Acquire a read lock

`void` **ObtainReadLock**`(IN struct Lock *lock)`

Description

This macro obtains a read lock on the specified `lock` object. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Read locks are incompatible with write, shared, and boosted shared locks.

Error Codes

--- No value is returned.

2.3.2.3 ObtainWriteLock — Acquire a write lock

`void` **ObtainWriteLock**`(IN struct Lock *lock)`

Description

This macro obtains a write lock on the specified *lock* object. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Write locks are incompatible with all other locks.

Error Codes

`---` No value is returned.

2.3.2.4 ObtainSharedLock — Acquire a shared lock

`void` **ObtainSharedLock**`(IN struct Lock *lock)`

Description

This macro obtains a shared lock on the specified *lock* object. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Shared locks are incompatible with write and boosted shared locks, but are compatible with read locks.

Error Codes

`---` No value is returned.

2.3.2.5 **ReleaseReadLock** — Release read lock

`void` **ReleaseReadLock**`(IN struct Lock *lock)`

Description

This macro releases the specified `lock`. The `lock` must have been previously read-locked. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation. The results are also unpredictable if the lock was not previously read-locked by the thread calling *ReleaseReadLock()*.

Error Codes

`---` No value is returned.

2.3.2.6 **ReleaseWriteLock** — Release write lock

`void` **ReleaseWriteLock**`(IN struct Lock *lock)`

Description

This macro releases the specified `lock`. The `lock` must have been previously write-locked. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation. The results are also unpredictable if the lock was not previously write-locked by the thread calling *ReleaseWriteLock()*.

Error Codes

`---` No value is returned.

2.3.2.7 ReleaseSharedLock — Release shared lock

`void` **ReleaseSharedLock**`(IN struct Lock *lock)`

Description

This macro releases the specified `lock`. The `lock` must have been previously share-locked. Since this is a macro and not a function call, results are not predictalbe if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation. The results are also unpredictable if the lock was not previously share-locked by the thread calling *ReleaseSharedLock()*.

Error Codes

`---` No value is returned.

2.3.2.8 CheckLock — Determine state of a lock

`void` **CheckLock**`(IN struct Lock *lock)`

Description

This macro produces an integer that specifies the status of the indicated lock. The value will be -1 if the lock is write-locked, 0 if unlocked, or otherwise a positive integer that indicates the number of readers (threads holding read locks). Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Error Codes

`---` No value is returned.

2.3.2.9 BoostLock — Boost a shared lock

`void` **BoostLock**`(IN struct Lock *lock)`

Description

This macro promotes ("boosts") a shared lock into a write lock. Such a boost operation guarantees that no other writer can get into the critical section in the process. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Error Codes

`---` No value is returned.

2.3.2.10 UnboostLock — Unboost a shared lock

`void` **UnboostLock**`(IN struct Lock *lock)`

Description

This macro demotes a boosted shared lock back down into a regular shared lock. Such an unboost operation guarantees that no other writer can get into the critical section in the process. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Error Codes

`---` No value is returned.

2.3.3 IOMGR

This section covers the calling interfaces to the I/O management package.

2.3.3.1 IOMGR_Initialize — Initialize the package

`int` **IOMGR_Initialize**`()`

Description

This function initializes the *IOMGR* package. Its main task is to create the *IOMGR* thread itself, which runs at the lowest possible priority (0). The remainder of the lightweight processes must be running at priority 1 or greater (up to a maximum of `LWP_MAX_PRIORITY` (4)) for the *IOMGR* package to function correctly.

Error Codes

`-1` The *LWP* and/or timer package haven't been initialized.

`<misc>` Any errors that may be returned by the *LWP_CreateProcess()* routine.

2.3.3.2 IOMGR_Finalize — Clean up the IOMGR facility

```
int IOMGR_Finalize()
```

Description

This routine cleans up after the *IOMGR* package when it is no longer needed. It releases all storage and destroys the *IOMGR* thread itself.

Error Codes

`<misc>` Any errors that may be returned by the *LWP_DestroyProcess()* routine.

2.3.3.3 IOMGR_Select — Perform a thread-level *select()*

```
int IOMGR_Select(IN int numfds;
                 IN int *rfds;
                 IN int *wfds;
                 IN int *xfds;
                 IN struct timeval *timeout)
```

Description

This routine performs an *LWP* version of UNIX *select()* operation. The parameters have the same meanings as with the UNIX call. However, the return values will be simplified (see below). If this is a polling select (i.e., the value of `timeout` is null), it is done and the *IOMGR_Select()* function returns to the user with the results. Otherwise, the calling thread is put to sleep. If at some point the *IOMGR* thread is the only runnable process, it will awaken and collect all select requests. The *IOMGR* will then perform a single select and awaken the appropriate processes. This will force a return from the affected *IOMGR_Select()* calls.

Error Codes

`-1` An error occurred.

`0` A timeout occurred.

`1` Some number of file descriptors are ready.

2.3.3.4 IOMGR_Signal — Associate UNIX and LWP signals

```
int IOMGR_Signal(IN int signo;
                 IN char *event)
```

Description

This function associates an *LWP* signal with a UNIX signal. After this call, when the given UNIX signal `signo` is delivered to the (heavyweight UNIX) process, the *IOMGR* thread will deliver an *LWP* signal to the `event` via *LWP_NoYieldSignal()*. This wakes up any lightweight processes waiting on the `event`. Multiple deliveries of the signal may be coalesced into one *LWP* wakeup. The call to *LWP_NoYieldSignal()* will happen synchronously. It is safe for an LWP to check for some condition and then go to sleep waiting for a UNIX signal without having to worry about delivery of the signal happening between the check and the call to *LWP_WaitProcess()*.

Error Codes

`LWP_EBADSIG` The `signo` value is out of range.

`LWP_EBADEVENT` The `event` pointer is null.

2.3.3.5 IOMGR_CancelSignal — Cancel UNIX and LWP signal association

int **IOMGR_CancelSignal**(IN int signo)

Description

This routine cancels the association between a UNIX signal and an *LWP* event. After calling this function, the UNIX signal `signo` will be handled however it was handled before the corresponding call to *IOMGR_Signal()*.

Error Codes

`LWP_EBADSIG` The `signo` value is out of range.

2.3.3.6 IOMGR_Sleep — Sleep for a given period

void **IOMGR_Sleep**(IN unsigned seconds)

Description

This function calls *IOMGR_Select()* with zero file descriptors and a timeout structure set up to cause the thread to sleep for the given number of `seconds`.

Error Codes

`---` No value is returned.

2.3.4 Timer

This section covers the calling interface to the timer package associated with the *LWP* facility.

2.3.4.1 TM_Init — Initialize a timer list

`int` **TM_Init**`(IN struct TM_Elem **list)`

Description

This function causes the specified timer `list` to be initialized. *TM_Init()* must be called before any other timer operations are applied to the list.

Error Codes

`-1` A null timer list could not be produced.

2.3.4.2 TM_Final — Clean up a timer list

`int` **TM_Final**`(IN struct TM_Elem **list)`

Description

This routine is called when the given empty timer `list` is no longer needed. All storage associated with the list is released.

Error Codes

`-1` The `list` parameter is invalid.

2.3.4.3 TM_Insert — Insert an object into a timer list

```
void TM_Insert(IN struct TM_Elem **list;
               IN struct TM_Elem *elem)
```

Description

This routine enters an new element, `elem`, into the list denoted by `list`. Before the new element is queued, its `TimeLeft` field (the amount of time before the object comes due) is set to the value stored in its `TotalTime` field. In order to keep `TimeLeft` fields current, the *TM_Rescan()* function may be used.

Error Codes

`---` No return value is generated.

2.3.4.4 TM_Rescan — Update all timers in the list

```
int TM_Rescan(IN struct TM_Elem *list)
```

Description

This function updates the `TimeLeft` fields of all timers on the given `list`. This is done by checking the time-of-day clock. Note: this is the only routine other than *TM_Init()* that updates the `TimeLeft` field in the elements on the list.

Instead of returning a value indicating success or failure, *TM_Rescan()* returns the number of entries that were discovered to have timed out.

Error Codes

`---` Instead of error codes, the number of entries that were discovered to have timed out is returned.

2.3.4.5 TM_GetExpired — Returns an expired timer

`struct TM_Elem *`**TM_GetExpired**`(IN struct TM_Elem *list)`

Description

This routine searches the specified timer `list` and returns a pointer to an expired timer element from that list. An expired timer is one whose `TimeLeft` field is less than or equal to zero. If there are no expired timers, a null element pointer is returned.

Error Codes

`---` Instead of error codes, an expired timer pointer is returned, or a null timer pointer if there are no expired timer objects.

2.3.4.6 TM_GetEarliest — Returns earliest unexpired timer

```
struct TM_Elem *TM_GetEarliest(IN struct TM_Elem *list)
```

Description

This function returns a pointer to the timer element that will be next to expire on the given `list`. This is defined to be the timer element with the smallest (positive) `TimeLeft` field. If there are no timers on the list, or if they are all expired, this function will return a null pointer.

Error Codes

`---` Instead of error codes, a pointer to the next timer element to expireis returned, or a null timer object pointer if they are all expired.

2.3.4.7 TM_eql — Test for equality of two timestamps

```
bool TM_eql(IN struct timemval *t1;
            IN struct timemval *t2)
```

Description

This function compares the given timestamps, `t1` and `t2`, for equality. Note that the function return value, `bool`, has been set via typedef to be equivalent to `unsigned char`.

Error Codes

`0` If the two timestamps differ.

`1` If the two timestamps are identical.

2.3.5 Fast Time

This section covers the calling interface to the fast time package associated with the *LWP* facility.

2.3.5.1 FT_Init — Initialize the fast time package

```
int FT_Init(IN int printErrors;
            IN int notReally)
```

Description

This routine initializes the fast time package, mapping in the kernel page containing the time-of-day variable. The `printErrors` argument, if non-zero, will cause any errors in initalization to be printed to `stderr`. The `notReally` parameter specifies whether initialization is really to be done. Other calls in this package will do auto-initialization, and hence the option is offered here.

Error Codes

`-1` Indicates that future calls to *FT_GetTimeOfDay()* will still work, but will not be able to access the information directly, having to make a kernel call every time.

2.3.5.2 FT_GetTimeOfDay — Initialize the fast time package

```
int FT_GetTimeOfDay(IN struct timeval *tv;
                    IN struct timezone *tz)
```

Description

This routine is meant to mimic the parameters and behavior of the UNIX *gettimeofday()* function. However, as implemented, it simply calls *gettimeofday()* and then does some bound-checking to make sure the value is reasonable.

Error Codes

`<misc>` Whatever value was returned by *gettimeofday()* internally.

2.3.6 Preemption

This section covers the calling interface to the preemption package associated with the *LWP* facility.

2.3.6.1 PRE_InitPreempt — Initialize the preemption package

`int` **PRE_InitPreempt**`(IN struct timeval *slice)`

Description

This function must be called to initialize the preemption package. It must appear sometime after the call to *LWP_InitializeProcessSupport()* and sometime before the first call to any other preemption routine. The `slice` argument specifies the time slice size to use. If the `slice` pointer is set to null in the call, then the default time slice, `DEFAULTSLICE` (10 milliseconds), will be used. This routine uses the UNIX interval timer and handling of the UNIX alarm signal, `SIGALRM`, to implement this timeslicing.

Error Codes

`LWP_EINIT` The *LWP* package hasn't been initialized.

`LWP_ESYSTEM` Operations on the signal vector or the interval timer have failed.

2.3.6.2 PRE_EndPreempt — Finalize the preemption package

`int` **PRE_EndPreempt**`()`

Description

This routine finalizes use of the preemption package. No further preemptions will be made. Note that it is not necessary to make this call before exit. *PRE_EndPreempt()* is provided only for those applications that wish to continue after turning off preemption.

Error Codes

`LWP_EINIT` The *LWP* package hasn't been initialized.

`LWP_ESYSTEM` Operations on the signal vector or the interval timer have failed.

2.3.6.3 PRE_PreemptMe — Mark thread as preemptible

`int` **PRE_PreemptMe**`()`

Description

This macro is used to signify the current thread as a candidate for preemption. The *LWP_InitializeProcessSupport()* routine must have been called before *PRE_PreemptMe()*.

Error Codes

`---` No return code is generated.

2.3.6.4 PRE_BeginCritical — Enter thread critical section

`int` **PRE_BeginCritical**`()`

Description

This macro places the current thread in a critical section. Upon return, and for as long as the thread is in the critical section, involuntary preemptions of this LWP will no longer occur.

Error Codes

`---` No return code is generated.

2.3.6.5 PRE_EndCritical — Exit thread critical section

`int` **PRE_EndCritical**`()`

Description

This macro causes the executing thread to leave a critical section previously entered via *PRE_BeginCritical()*. If involuntary preemptions were possible before the matching *PRE_BeginCritical()*, they are once again possible.

Error Codes

`---` No return code is generated.

Chapter 3

Rxkad

3.1 Introduction

The `rxkad` security module is offered as one of the built-in *Rx* authentication models. It is based on the Kerberos system developed by MIT's Project Athena. Readers wishing detailed information regarding Kerberos design and implementation are directed to [6]. This chapter is devoted to defining how Kerberos authentication services are made available as *Rx* components, and assumes the reader has some familiarity with Kerberos. Included are descriptions of how client-side and server-side *Rx* security objects (`struct rx_securityClass`; see Section 5.3.1.1) implementing this protocol may be generated by an *Rx* application. Also, a description appears of the set of routines available in the associated `struct rx_securityOps` structures, as covered in Section 5.3.1.2. It is strongly recommended that the reader become familiar with this section on `struct rx_securityOps` before reading on.

3.2 Definitions

An important set of definitions related to the *rxkad* security package is provided by the *rxkad.h* include file. Determined here are various values for ticket lifetimes, along with structures for encryption keys and Kerberos principals. Declarations for the two routines required to generate the different *rxkad* security objects also appear here. The two functions are named *rxkad_NewServerSecurityObject()* and *rxkad_NewClientSecurityObject()*. In addition, type field values, encryption levels, security index operations, and statistics structures may be found in this file.

3.3 Exported Objects

To be usable as an *Rx* security module, the *rxkad* facility exports routines to create server-side and client-side security objects. The server authentication object is incorporated into the server code when calling *rx_NewService()*. The client authentication object is incorporated into the client code every time a connection is established via *rx_NewConnection()*. Also, in order to implement these security objects, the *rxkad* module must provide definitions for some subset of the generic security operations as defined in the appropriate `struct rx_securityOps` variable.

3.3.1 Server-Side Mechanisms

3.3.1.1 Security Operations

The server side of the *rxkad* module fills in all but two of the possible routines associated with an *Rx* security object, as described in Section 5.3.1.2.

```
static struct rx_securityOps rxkad_server_ops = {
    rxkad_Close,
    rxkad_NewConnection,
    rxkad_PreparePacket,            /*Once per packet creation*/
    0,                              /*Send packet (once per retrans)*/
    rxkad_CheckAuthentication,
    rxkad_CreateChallenge,
    rxkad_GetChallenge,
    0,
    rxkad_CheckResponse,
    rxkad_CheckPacket,              /*Check data packet*/
    rxkad_DestroyConnection,
    rxkad_GetStats,
};
```

The *rxkad* service does not need to take any special action each time a packet belonging to a call in an *rxkad Rx* connection is physically transmitted. Thus, a routine is not supplied for the *op_SendPacket()* function slot. Similarly, no preparatory work needs to be done previous to the reception of a response packet from a security challenge, so the *op_GetResponse()* function slot is also empty.

3.3.1.2 Security Object

The exported routine used to generate an *rxkad*-specific server-side security class object is named *rxdad_NewServerSecurityObject()*. It is declared with four parameters, as follows:

```
struct rx_securityClass *
rxkad_NewServerSecurityObject(a_level, a_getKeyRockP, a_getKeyP, a_userOKP)
  rxkad_level    a_level;       /*Minimum level*/
  char          *a_getKeyRockP; /*Rock for get_key implementor*/
  int           (*a_getKeyP)(); /*Passed kvno & addr(key) to fill*/
  int           (*a_userOKP)(); /*Passed name, inst, cell => bool*/
```

The first argument specifies the desired level of encryption, and may take on the following values (as defined in *rxkad.h*):

- `rxkad_clear`: Specifies that packets are to be sent entirely in the clear, without any encryption whatsoever.
- `rxkad_auth`: Specifies that packet sequence numbers are to be encrypted.
- `rxkad_crypt`: Specifies that the entire data packet is to be encrypted.

The second and third parameters represent, respectively, a pointer to a private data area, sometimes called a "rock", and a procedure reference that is called with the key version number accompanying the Kerberos ticket and returns a pointer to the server's decryption key. The fourth argument, if not null, is a pointer to a function that will be called for every new connection with the client's name, instance, and cell. This routine should return zero if the user is *not* acceptable to the server.

3.3.2 Client-Side Mechanisms

3.3.2.1 Security Operations

The client side of the *rxkad* module fills in relatively few of the routines associated with an *Rx* security object, as demonstrated below. The general *Rx* security object, of which this is an instance, is described in detail in Section 5.3.1.2.

```
static struct rx_securityOps rxkad_client_ops = {
    rxkad_Close,
```

```
    rxkad_NewConnection,                /*Every new connection*/
    rxkad_PreparePacket,                /*Once per packet creation*/
    0,                                  /*Send packet (once per retrans)*/
    0,
    0,
    0,
    rxkad_GetResponse,                  /*Respond to challenge packet*/
    0,
    rxkad_CheckPacket,                  /*Check data packet*/
    rxkad_DestroyConnection,
    rxkad_GetStats,
    0,
    0,
    0,
};
```

As expected, routines are defined for use when someone destroys a security object (*rxkad_Close()*) and when an *Rx* connection using the *rxkad* model creates a new connection (*rxkad_NewConnection()*) or deletes an existing one (*rxkad_DestroyConnection()*). Security-specific operations must also be performed in behalf of *rxkad* when packets are created (*rxkad_PreparePacket()*) and received (*rxkad_CheckPacket()*). Finally, the client side of an *rxkad* security object must also be capable of constructing responses to security challenges from the server (*rxkad_GetResponse()*) and be willing to reveal statistics on its own operation (*rxkad_GetStats()*).

3.3.2.2 Security Object

The exported routine used to generate an *rxkad*-specific client-side security class object is named *rxkad_NewClientSecurityObject()*. It is declared with five parameters, specified below:

```
struct rx_securityClass *
rxkad_NewClientSecurityObject(a_level, a_sessionKeyP, a_kvno,
                              a_ticketLen, a_ticketP)
  rxkad_level              a_level;
  struct ktc_encryptionKey *a_sessionKeyP;
  long                     a_kvno;
  int                      a_ticketLen;
  char                    *a_ticketP;
```

The first parameter, `a_level`, specifies the level of encryption desired for this security object, with legal choices being identical to those defined for the server-side security object described in Section 3.3.1.2. The second parameter, `a_sessionKeyP`, provides the session key to use. The `ktc_encryptionKey` structure is defined in the *rxkad.h* include file, and consists of an array of 8 characters. The third parameter, `a_kvno`, provides the key version number associated with `a_sessionKeyP`. The fourth argument, `a_ticketLen`, communicates the length in bytes of the data stored in the fifth parameter, `a_ticketP`, which points to the Kerberos ticket to use for the principal for which the security object will operate.

Chapter 4

Rx **Support Packages**

4.1 Introduction

This chapter documents three packages defined directly in support of the *Rx* facility.

1. **rx_queue**: Doubly-linked queue package.
2. **rx_clock**: Clock package, using the 4.3BSD interval timer.
3. **rx_event**: Future events package.

References to constants, structures, and functions defined by these support packages will appear in the following API chapter.

4.2 The *rx_queue* Package

This package provides a doubly-linked queue structure, along with a full suite of related operations. The main concern behind the coding of this facility was efficiency. All functions are implemented as macros, and it is suggested that only simple expressions be used for all parameters.

The *rx_queue* facility is defined by the *rx_queue.h* include file. Some macros visible in this file are intended for *rx_queue* internal use only. An understanding of these "hidden" macros is important, so they will also be described by this document.

4.2.1 struct queue

The `queue` structure provides the linkage information required to maintain a queue of objects. The `queue` structure is prepended to any user-defined data type which is to be organized in this fashion.

Fields

struct queue *prev - Pointer to the previous queue header.

struct queue *next - Pointer to the next queue header.

Note that a null *Rx* queue consists of a single `struct queue` object whose next and previous pointers refer to itself.

4.2.2 Internal Operations

This section describes the internal operations defined for *Rx* queues. They will be referenced by the external operations documented in Section 4.2.3.

4.2.2.1 *_Q()*: Coerce type to a queue element

```
#define _Q(x) ((struct queue *)(x))
```

This operation coerces the user structure named by `x` to a queue element. Any user structure using the *rx_queue* package must have a `struct queue` as its first field.

4.2.2.2 *_QA()*: Add a queue element before/after another element

```
#define _QA(q,i,a,b) (((i->a=q->a)->b=i)->b=q, q->a=i)
```

This operation adds the queue element referenced by `i` either before or after a queue element represented by `q`. If the (`a, b`) argument pair corresponds to an element's (`next, prev`) fields, the new element at `i` will be linked after `q`. If the (`a, b`) argument pair corresponds to an element's (`prev, next`) fields, the new element at `i` will be linked before `q`.

4.2.2.3 *_QR()*: Remove a queue element

```
#define _QR(i) ((_Q(i)->prev->next=_Q(i)->next)->prev=_Q(i)->prev)
```

This operation removes the queue element referenced by `i` from its queue. The `prev` and `next` fields within queue element `i` itself is *not* updated to reflect the fact that it is no longer part of the queue.

4.2.2.4 *_QS()*: Splice two queues together

```
#define _QS(q1,q2,a,b) if (queue_IsEmpty(q2)); else
    ((((q2->a->b=q1)->a->b=q2->b)->a=q1->a, q1->a=q2->a),
     queue_Init(q2))
```

This operation takes the queues identified by `q1` and `q2` and splices them together into a single queue. The order in which the two queues are appended is determined by the `a` and `b` arguments. If the (`a, b`) argument pair corresponds to `q1`'s (`next, prev`) fields, then `q2` is appended to `q1`. If the (`a, b`) argument pair corresponds to `q1`'s (`prev, next`) fields, then `q` is prepended to `q2`.

This internal *_QS()* routine uses two exported queue operations, namely *queue_Init()* and *queue_IsEmpty()*, defined in Sections 4.2.3.1 and 4.2.3.16 respectively below.

4.2.3 External Operations

4.2.3.1 *queue_Init()*: Initialize a queue header

```
#define queue_Init(q) (_Q(q))->prev = (_Q(q))->next = (_Q(q))
```

The queue header referred to by the `q` argument is initialized so that it describes a null (empty) queue. A queue head is simply a queue element.

4.2.3.2 *queue_Prepend()*: Put element at the head of a queue

```
#define queue_Prepend(q,i) _QA(_Q(q),_Q(i),next,prev)
```

Place queue element `i` at the head of the queue denoted by `q`. The new queue element, `i`, should not currently be on any queue.

4.2.3.3 *queue_Append()*: Put an element a the tail of a queue

```
#define queue_Append(q,i) _QA(_Q(q),_Q(i),prev,next)
```

Place queue element `i` at the tail of the queue denoted by `q`. The new queue element, `i`, should not currently be on any queue.

4.2.3.4 *queue_InsertBefore()*: Insert a queue element before another element

```
#define queue_InsertBefore(i1,i2) _QA(_Q(i1),_Q(i2),prev,next)
```

Insert queue element `i2` before element `i1` in `i1`'s queue. The new queue element, `i2`, should not currently be on any queue.

4.2.3.5 *queue_InsertAfter()*: Insert a queue element after another element

```
#define queue_InsertAfter(i1,i2) _QA(_Q(i1),_Q(i2),next,prev)
```

Insert queue element `i2` after element `i1` in `i1`'s queue. The new queue element, `i2`, should not currently be on any queue.

4.2.3.6 *queue_SplicePrepend()*: Splice one queue before another

```
#define queue_SplicePrepend(q1,q2) _QS(_Q(q1),_Q(q2),next,prev)
```

Splice the members of the queue located at `q2` to the beginning of the queue located at `q1`, reinitializing queue `q2`.

4.2.3.7 *queue_SpliceAppend()*: Splice one queue after another

```
#define queue_SpliceAppend(q1,q2) _QS(_Q(q1),_Q(q2),prev,next)
```

Splice the members of the queue located at `q2` to the end of the queue located at `q1`, reinitializing queue `q2`. Note that the implementation of *queue_SpliceAppend()* is identical to that of *queue_SplicePrepend()* except for the order of the `next` and `prev` arguments to the internal queue splicer, *_QS()*.

4.2.3.8 *queue_Replace()*: Replace the contents of a queue with that of another

```
#define queue_Replace(q1,q2) (*_Q(q1) = *_Q(q2),
            _Q(q1)->next->prev = _Q(q1)->prev->next = _Q(q1),
            queue_Init(q2))
```

Replace the contents of the queue located at `q1` with the contents of the queue located at `q2`. The `prev` and `next` fields from `q2` are copied into the queue object referenced by `q1`, and the appropriate element pointers are reassigned. After the replacement has occurred, the queue header at `q2` is reinitialized.

4.2.3.9 *queue_Remove()*: Remove an element from its queue

```
#define queue_Remove(i) (_QR(i), _Q(i)->next = 0)
```

This function removes the queue element located at `i` from its queue. The `next` field for the removed entry is zeroed. Note that multiple removals of the same queue item are not supported.

4.2.3.10 *queue_MoveAppend()*: Move an element from its queue to the end of another queue

```
#define queue_MoveAppend(q,i) (_QR(i), queue_Append(q,i))
```

This macro removes the queue element located at `i` from its current queue. Once removed, the element at `i` is appended to the end of the queue located at `q`.

4.2.3.11 *queue_MovePrepend()*: Move an element from its queue to the head of another queue

```
#define queue_MovePrepend(q,i) (_QR(i), queue_Prepend(q,i))
```

This macro removes the queue element located at `i` from its current queue. Once removed, the element at `i` is inserted at the head fo the queue located at `q`.

4.2.3.12 *queue_First()*: Return the first element of a queue, coerced to a particular type

```
#define queue_First(q,s) ((struct s *)_Q(q)->next)
```

Return a pointer to the first element of the queue located at `q`. The returned pointer value is coerced to conform to the given `s` structure. Note that a properly coerced pointer to the queue head is returned if `q` is empty.

4.2.3.13 *queue_Last()*: Return the last element of a queue, coerced to a particular type

```
#define queue_Last(q,s) ((struct s *)_Q(q)->prev)
```

Return a pointer to the last element of the queue located at `q`. The returned pointer value is coerced to conform to the given `s` structure. Note that a properly coerced pointer to the queue head is returned if `q` is empty.

4.2.3.14 *queue_Next()*: Return the next element of a queue, coerced to a particular type

```
#define queue_Next(i,s) ((struct s *)_Q(i)->next)
```

Return a pointer to the queue element occuring after the element located at `i`. The returned pointer value is coerced to conform to the given `s` structure. Note that a properly coerced pointer to the queue head is returned if item `i` is the last in its queue.

4.2.3.15 *queue_Prev()*: Return the next element of a queue, coerced to a particular type

```
#define queue_Prev(i,s) ((struct s *)_Q(i)->prev)
```

Return a pointer to the queue element occuring before the element located at `i`. The returned pointer value is coerced to conform to the given `s` structure. Note that a properly coerced pointer to the queue head is returned if item `i` is the first in its queue.

4.2.3.16 *queue_IsEmpty()*: Is the given queue empty?

```
#define queue_IsEmpty(q) (_Q(q)->next == _Q(q))
```

Return a non-zero value if the queue located at `q` does not have any elements in it. In this case, the queue consists solely of the queue header at `q` whose `next` and `prev` fields reference itself.

4.2.3.17 *queue_IsNotEmpty()*: Is the given queue not empty?

```
#define queue_IsNotEmpty(q) (_Q(q)->next != _Q(q))
```

Return a non-zero value if the queue located at `q` has at least one element in it other than the queue header itself.

4.2.3.18 *queue_IsOnQueue()*: Is an element currently queued?

```
#define queue_IsOnQueue(i) (_Q(i)->next != 0)
```

This macro returns a non-zero value if the queue item located at `i` is currently a member of a queue. This is determined by examining its `next` field. If it is non-null, the element is considered to be queued. Note that any element operated on by *queue_Remove()* (Section 4.2.3.9) will have had its `next` field zeroed. Hence, it would cause a non-zero return from this call.

4.2.3.19 *queue_IsFirst()*: Is an element the first on a queue?

```
#define queue_IsFirst(q,i) (_Q(q)->first == _Q(i))
```

This macro returns a non-zero value if the queue item located at `i` is the first element in the queue denoted by `q`.

4.2.3.20 *queue_IsLast()*: Is an element the last on a queue?

```
#define queue_IsLast(q,i) (_Q(q)->prev == _Q(i))
```

This macro returns a non-zero value if the queue item located at `i` is the last element in the queue denoted by `q`.

4.2.3.21 *queue_IsEnd()*: Is an element the end of a queue?

```
#define queue_IsEnd(q,i) (_Q(q) == _Q(i))
```

This macro returns a non-zero value if the queue item located at `i` is the end of the queue located at `q`. Basically, it determines whether a queue element in question is also the queue header structure itself, and thus does not represent an actual queue element. This function is useful for terminating an iterative sweep through a queue, identifying when the search has wrapped to the queue header.

4.2.3.22 *queue_Scan()*: `for` loop test for scanning a queue in a forward direction

```
#define queue_Scan(q, qe, next, s)
    (qe) = queue_First(q, s), next = queue_Next(qe, s);
    !queue_IsEnd(q, qe);
    (qe) = (next), next = queue_Next(qe, s)
```

This macro may be used as the body of a `for` loop test intended to scan through each element in the queue located at `q`. The `qe` argument is used as the `for` loop variable. The `next` argument is used to store the next value for `qe` in the upcoming loop iteration. The `s` argument provides the name of the structure to which each queue element is to be coerced. Thus, the values provided for the `qe` and `next` arguments must be of type `(struct s *)`.

An example of how *queue_Scan()* may be used appears in the code fragment below. It declares a structure named `mystruct`, which is suitable for queueing. This queueable structure is composed of the queue pointers themselves followed by an integer value. The actual queue header is kept in `demoQueue`, and the `currItemP` and `nextItemP` variables are used to step through the `demoQueue`. The *queue_Scan()* macro is used in the `for` loop to generate references in `currItemP` to each queue element in turn for each iteration. The loop is used to increment every queued structure's `myval` field by one.

```
struct mystruct {
    struct queue q;
    int myval;
};
```

```
struct queue demoQueue;
struct mystruct *currItemP, *nextItemP;

...

for (queue_Scan(&demoQueue, currItemP, nextItemP, mystruct)) {
    currItemP->myval++;
}
```

Note that extra initializers can be added before the body of the *queue_Scan()* invocation above, and extra expressions can be added afterwards.

4.2.3.23 *queue_ScanBackwards()*: `for` loop test for scanning a queue in a reverse direction

```
#define queue_ScanBackwards(q, qe, prev, s)
    (qe) = queue_Last(q, s), prev = queue_Prev(qe, s);
    !queue_IsEnd(q, qe);
    (qe) = prev, prev = queue_Prev(qe, s)
```

This macro is identical to the *queue_Scan()* macro described above in Section 4.2.3.22 except for the fact that the given queue is scanned backwards, starting at the last item in the queue.

4.3 The *rx_clock* Package

This package maintains a clock which is independent of the time of day. It uses the UNIX 4.3BSD interval timer (e.g., *getitimer()*, *setitimer()*) in `TIMER_REAL` mode. Its definition and interface may be found in the *rx_clock.h* include file.

4.3.1 `struct clock`

This structure is used to represent a clock value as understood by this package. It consists of two fields, storing the number of seconds and microseconds that have elapsed since the associated `clock_Init()` routine has been called.

Fields

long sec - Seconds since call to *clock_Init()*.

long usec - Microseconds since call to *clock_Init()*.

4.3.2 `clock_nUpdates`

The integer-valued `clock_nUpdates` is a variable exported by the *rx_clock* facility. It records the number of times the clock value is actually updated. It is bumped each time the *clock_UpdateTime()* routine is called, as described in Section 4.3.3.2.

4.3.3 Operations

4.3.3.1 *clock_Init()*: Initialize the clock package

This routine uses the UNIX *setitimer()* call to initialize the UNIX interval timer. If the *setitimer()* call fails, an error message will appear on `stderr`, and an *exit(1)* will be executed.

4.3.3.2 *clock_UpdateTime()*: Compute the current time

The *clock_UpdateTime()* function calls the UNIX *getitimer()* routine in order to update the current time. The exported `clock_nUpdates` variable is incremented each time the *clock_UpdateTime()* routine is called.

4.3.3.3 *clock_GetTime()*: Return the current clock time

This macro updates the current time if necessary, and returns the current time into the `cv` argument, which is declared to be of type `(struct clock *)`.

4.3.3.4 *clock_Sec()*: Get the current clock time, truncated to seconds

This macro returns the `long` value of the `sec` field of the current time. The recorded time is updated if necessary before the above value is returned.

4.3.3.5 *clock_ElapsedTime()*: Measure milliseconds between two given clock values

This macro returns the elapsed time in milliseconds between the two clock structure pointers provided as arguments, `cv1` and `cv2`.

4.3.3.6 *clock_Advance()*: Advance the recorded clock time by a specified clock value

This macro takes a single `(struct clock *)` pointer argument, `cv`, and adds this clock value to the internal clock value maintined by the package.

4.3.3.7 *clock_Gt()*: Is a clock value greater than another?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the `a` parameter points to a clock value which is later than the one pointed to by `b`.

4.3.3.8 *clock_Ge()*: Is a clock value greater than or equal to another?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the `a` parameter points to a clock value which is greater than or equal to the one pointed to by `b`.

4.3.3.9 *clock_Gt()*: Are two clock values equal?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the clock values pointed to by `a` and `b` are equal.

4.3.3.10 *clock_Le()*: Is a clock value less than or equal to another?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the `a` parameter points to a clock value which is less than or equal to the one pointed to by `b`.

4.3.3.11 *clock_Lt()*: Is a clock value less than another?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the `a` parameter points to a clock value which is less than the one pointed to by `b`.

4.3.3.12 *clock_IsZero()*: Is a clock value zero?

This macro takes a single parameter of type `(struct clock *)`, `c`. It returns a non-zero value if the `c` parameter points to a clock value which is equal to zero.

4.3.3.13 *clock_Zero()*: Set a clock value to zero

This macro takes a single parameter of type `(struct clock *)`, `c`. It sets the given clock value to zero.

4.3.3.14 *clock_Add()*: Add two clock values together

This macro takes two parameters of type `(struct clock *)`, `c1` and `c2`. It adds the value of the time in `c2` to `c1`. Both clock values must be positive.

4.3.3.15 *clock_Sub()*: Subtract two clock values

This macro takes two parameters of type `(struct clock *)`, `c1` and `c2`. It subtracts the value of the time in `c2` from `c1`. The time pointed to by `c2` should be less than the time pointed to by `c1`.

4.3.3.16 *clock_Float()*: Convert a clock time into floating point

This macro takes a single parameter of type `(struct clock *)`, `c`. It expresses the given clock value as a floating point number.

4.4 The *rx_event* Package

This package maintains an event facility. An *event* is defined to be something that happens at or after a specified clock time, unless cancelled prematurely. The clock times used are those provided by the *rx_clock* facility described in Section 4.3 above. A user routine associated with an event is called with the appropriate arguments when that event occurs. There are some restrictions on user routines associated with such events. First, this user-supplied routine should *not* cause process preemption. Also, the event passed to the user routine is still resident on the event queue at the time of invocation. The user must not remove this event explicitly (via an *event_Cancel()*, see below). Rather, the user routine may remove or schedule any *other* event at this time.

The events recorded by this package are kept queued in order of expiration time, so that the first entry in the queue corresponds to the event which is the first to expire. This interface is defined by the *rx_event.h* include file.

4.4.1 `struct rxevent`

This structure defines the format of an *Rx* event record.

Fields

struct queue junk - The queue to which this event belongs.

struct clock eventTime - The clock time recording when this event comes due.

int (*func)() - The user-supplied function to call upon expiration.

char *arg - The first argument to the *(*func)()* function above.

char *arg1 - The second argument to the *(*func)()* function above.

4.4.2 Operations

This section covers the interface routines provided for the *Rx* event package.

4.4.2.1 *rxevent_Init()*: Initialize the event package

The *rxevent_Init()* routine takes two arguments. The first, `nEvents`, is an integer-valued parameter which specifies the number of event structures to allocate at one time. This specifies the appropriate granularity of memory allocation by the event package. The second parameter, `scheduler`, is a pointer to an integer-valued function. This function is to be called when an event is *posted* (added to the set of events managed by the package) that is scheduled to expire before any other existing event.

This routine sets up future event allocation block sizes, initializes the queues used to manage active and free event structures, and recalls that an initialization has occurred. Thus, this function may be safely called multiple times.

4.4.2.2 *rxevent_Post()*: Schedule an event

This function constructs a new event based on the information included in its parameters and then schedules it. The *rxevent_Post()* routine takes four parameters. The first is named `when`, and is of type `(struct clock *)`. It specifies the clock time at which the event is to occur. The second parameter is named `func` and is a pointer to the integer-valued function to associate with the event that will be created. When the event comes due, this function will be executed by the event package. The next two arguments to *rxevent_Post()* are named `arg` and `arg1`, and are both of type `(char *)`. They serve as the two arguments thath will be supplied to the `func` routine when the event comes due.

If the given event is set to take place before any other event currently posted, the `scheduler` routine established when the *rxevent_Init()* routine was called will be executed. This gives the application a chance to react to this new event in a reasonable way. One might expect that this `scheduler` routine will alter sleep times used by the application to make sure that it executes in time to handle the new event.

4.4.2.3 *rxevent_Cancel_1()*: Cancel an event (internal use)

This routine removes an event from the set managed by this package. It takes a single parameter named `ev` of type `(struct rxevent *)`. The `ev` argument identifies the pending event to be cancelled.

The *rxevent_Cancel_1()* routine should never be called directly. Rather, it should be accessed through the *rxevent_Cancel()* macro, described in Section 4.4.2.4 below.

4.4.2.4 *rxevent_Cancel()*: Cancel an event (external use)

This macro is the proper way to call the *rxevent_Cancel_1()* routine described in Section 4.4.2.3 above. Like *rxevent_Cancel_1()*, it takes a single argument. This `event_ptr` argument is of type `(struct rxevent *)`, and identifies the pending event to be cancelled. This macro first checks to see if `event_ptr` is null. If not, it calls *rxevent_Cancel_1()* to perform the real work. The `event_ptr` argument is zeroed after the cancellation operation completes.

4.4.2.5 *rxevent_RaiseEvents()*: Initialize the event package

This function processes all events that have expired relative to the current clock time maintained by the event package. Each qualifying event is removed from the queue in order, and its user-supplied routine (*func()*) is executed with the associated arguments.

The *rxevent_RaiseEvents()* routine takes a single output parameter named `next`, defined to be of type `(struct clock *)`. Upon completion of *rxevent_RaiseEvents()*, the relative time to the next event due to expire is placed in `next`. This knowledge may be used to calculate the amount of sleep time before more event processing is needed. If there is no recorded event which is still pending at this point, *rxevent_RaiseEvents()* returns a zeroed clock value into `next`.

4.4.2.6 *rxevent_TimeToNextEvent()*: Get amount of time until the next event expires

This function returns the time between the current clock value as maintained by the event package and the the next event's expiration time. This information is placed in the single output argument,`interval`, defined to be of type `(struct clock *)`. The *rxevent_TimeToNextEvent()* function returns integer-valued quantities. If there are no scheduled events, a zero is returned. If there are one or more scheduled events, a 1 is returned. If zero is returned, the `interval` argument is not updated.

Chapter 5

Programming Interface

5.1 Introduction

This chapter documents the API for the *Rx* facility. Included are descriptions of all the constants, structures, exported variables, macros, and interface functions available to the application programmer. This interface is identical regardless of whether the application lives within the UNIX kernel or above it.

This chapter actually provides more information than what may be strictly considered the *Rx* API. Many objects that were intended to be opaque and for *Rx* internal use only are also described here. The reason driving the inclusion of this "extra" information is that such exported *Rx* interface files as *rx.h* make these objects visible to application programmers. It is prefereable to describe these objects here than to ignore them and leave application programmers wondering as to their meaning.

An example application illustrating the use of this interface, showcasing code from both server and client sides, appears in the following chapter.

5.2 Constants

This section covers the basic constant definitions of interest to the *Rx* application programmer. Each subsection is devoted to describing the constants falling into the following categories:

- Configuration quantities

- Waiting options
- Connection ID operations
- Connection flags
- Connection types
- Call states
- Call flags
- Call modes
- Packet header flags
- Packet sizes
- Packet types
- Packet classes
- Conditions prompting ack packets
- Ack types
- Error codes
- Debugging values

An attempt has been made to relate these constant definitions to the objects or routines that utilize them.

5.2.1 Configuration Quantities

These definitions provide some basic *Rx* configuration parameters, including the number of simultaneous calls that may be handled on a single connection, lightweight thread parameters, and timeouts for various operations.

Name	*Value*	*Description*
RX_IDLE_DEAD_TIME	60	Default idle dead time for connections, in seconds.
RX_MAX_SERVICES	20	The maximum number of *Rx* services that may be installed within one application.
RX_PROCESS_MAXCALLS	4	The maximum number of asynchronous calls active simultaneously on any given *Rx* connection. This value must be set to a power of two.
RX_DEFAULT_STACK_SIZE	16,000	Default lightweight thread stack size, measured in bytes. This value may be overridden by calling the *rx_SetStackSize()* macro.
RX_PROCESS_PRIORITY	LWP_NORMAL_PRIORITY	This is the priority under which an *Rx* thread should run. There should not generally be any reason to change this setting.
RX_CHALLENGE_TIMEOUT	2	The number of seconds before another authentication request packet is generated
RX_MAXACKS	255	Maximum number of individual acknowledgements that may be carried in an *Rx* acknowledgement packet

5.2.2 Waiting Options

These definitions provide readable values indicating whether an operation should block when packet buffer resources are not available.

Name	*Value*	*Description*
RX_DONTWAIT	0	Wait until the associated operation completes
RX_WAIT	1	Don't wait if the associated operation would block

5.2.3 Connection ID Operations

These values assist in extracting the call channel number from a connection identifier. A call channel is the index of a particular asynchronous call structure within a single *Rx* connection.

Name	*Value*	*Description*
`RX_CIDSHIFT`	2	Number of bits to right-shift to isolate a connection ID. Must be set to the log (base two) of `RX_MAXCALLS`.
`RX_CHANNELMASK`	`(RX_MAXCALLS-1)`	Mask used to isolate a call channel from a connection ID field
`RX_CIDMASK`	`(R̃X_CHANNELMASK)`	Mask used to isolate the connection ID from its field, masking out the call channel information

5.2.4 Connection Flags

The values defined here appear in the `flags` field of *Rx* connections, as defined by the `rx_connection` structure described in Section 5.3.2.2.

Name	*Value*	*Description*
`RX_CONN_MAKECALL_WAITING`	1	*rx_MakeCall()* is waiting for a channel
`RX_CONN_DESTROY_ME`	2	Destroy this (client) connection after its last call completes
`RX_CONN_USING_PACKET_CKSUM`	4	This packet is using security-related check-summing (a non-zero header.spare field has been seen)

5.2.5 Connection Types

Rx stores different information in its connection structures, depending on whether the given connection represents the server side (the one providing the service) or the client side (the one requesting the service) of the protocol. The `type` field within the connection structure (described in Section 5.3.2.2) takes on the following values to differentiate the two types of connections, and identifies the fields that are active within the connection structure.

Name	*Value*	*Description*
`RX_CLIENT_CONNECTION`	0	This is a client-side connection.
`RX_SERVER_CONNECTION`	1	This is a server-side connection.

5.2.6 Call States

An *Rx* call on a particular connection may be in one of several states at any instant in time. The following definitions identify the range of states that a call may assume.

Name	*Value*	*Description*
`RX_STATE_NOTINIT`	0	The call structure has never been used, and is thus still completely uninitialized
`RX_STATE_PRECALL`	1	A call is not yet in progress, but packets have arrived for it anyway. This only applies to calls within server-side connections
`RX_STATE_ACTIVE`	2	This call is fully active, having an attached lightweight thread operating on its behalf
`RX_STATE_DALLY`	3	The call structure is "dallying" after its lightweight thread has completed its most recent call. This is a "hot-standby" condition, where the call structure preserves state from the previous call and thus optimizes the arrival of further, related calls.

5.2.7 Call Flags

These values are used within the `flags` field of a variable declared to be of type `struct rx_call`, as described in Section 5.3.2.4. They provide additional information as to the state of the given *Rx* call, such as the type of event for which it is waiting (if any) and whether or not all incoming packets have been received in support of the call.

Name	*Value*	*Description*
RX_CALL_READER_WAIT	1	Reader is waiting for next packet
RX_CALL_WAIT_WINDOW_ALLOC	2	Sender is waiting for a window so that it can allocate buffers
RX_CALL_WAIT_WINDOW_SEND	4	Sender is waiting for a window so that it can send buffers
RX_CALL_WAIT_PACKETS	8	Sender is waiting for packet buffers
RX_CALL_WAIT_PROC	16	The call is waiting for a lightweight thread to be assigned to the operation it has just received
RX_CALL_RECEIVE_DONE	32	All packets have been received on this call
RX_CALL_CLEARED	64	The receive queue has been cleared when in precall state

5.2.8 Call Modes

These values define the modes of an *Rx* call when it is in the `RX_STATE_ACTIVE` state, having a lightweight thread assigned to it.

Name	*Value*	*Description*
RX_MODE_SENDING	1	We are sending or ready to send
RX_MODE_RECEIVING	2	We are receiving or ready to receive
RX_MODE_ERROR	3	Something went wrong in the current conversation
RX_MODE_EOF	4	The server side has flushed (or the client side has read) the last reply packet

5.2.9 Packet Header Flags

Rx packets carry a flag field in their headers, providing additional information regarding the packet's contents. The *Rx* packet header's `flag` field's bits may take the following values:

Name	*Value*	*Description*
RX_CLIENT_INITIATED	1	Signifies that a packet has been sent/received from the client side of the call
RX_REQUEST_ACK	2	The *Rx* call's peer entity requests an acknowledgement
RX_LAST_PACKET	4	This is the final packet from this side of the call
RX_MORE_PACKETS	8	There are more packets following this, i.e. the next sequence number seen by the receiver should be greater than this one, rather than a retransmission of an earlier sequence number
RX_PRESET_FLAGS	(RX_CLIENT_INITIATED | RX_LAST_PACKET)	This flag is preset once per *Rx* packet. It doesn't change on retransmission of the packet

5.2.10 Packet Sizes

These values provide sizing information on the various regions within *Rx* packets. These packet sections include the IP/UDP headers and bodies as well *Rx* header and bodies. Also covered are such values as different maximum packet sizes depending on whether they are targeted to peers on the same local network or a more far-flung network. Note that the `MTU` term appearing below is an abbreviation for *Maximum Transmission Unit*.

Name	*Value*	*Description*
`RX_IPUDP_SIZE`	28	The number of bytes taken up by IP/UDP headers
`RX_MAX_PACKET_SIZE`	(1500 - RX_IPUDP_SIZE)	This is the Ethernet MTU minus IP and UDP header sizes
`RX_HEADER_SIZE`	sizeof (struct rx_header)	The number of bytes in an *Rx* packet header
`RX_MAX_PACKET_DATA_SIZE`	(RX_MAX_PACKET_SIZE - RX_HEADER_SIZE)	Maximum size in bytes of the user data in a packet
`RX_LOCAL_PACKET_SIZE`	RX_MAX_PACKET_SIZE	Packet size in bytes to use when being sent to a host on the same net.
`RX_REMOTE_PACKET_SIZE`	(576 - RX_IPUDP_SIZE)	Packet size in bytes to use when being sent to a host on a different net.

5.2.11 Packet Types

The following values are used in the `packetType` field within a `struct rx_packet`, and define the different roles assumed by *Rx* packets. These roles include user data packets, different flavors of acknowledgements, busies, aborts, authentication challenges and responses, and debugging vehicles.

Name	*Value*	*Description*
`RX_PACKET_TYPE_DATA`	1	A user data packet
`RX_PACKET_TYPE_ACK`	2	Acknowledgement packet
`RX_PACKET_TYPE_BUSY`	3	Busy packet. The server-side entity cannot accept the call at the moment, but the requestor is encouraged to try again later
`RX_PACKET_TYPE_ABORT`	4	Abort packet. No response is needed for this packet type
`RX_PACKET_TYPE_ACKALL`	5	Acknowledges receipt of all packets on a call
`RX_PACKET_TYPE_CHALLENGE`	6	Challenge the client's identity, requesting credentials
`RX_PACKET_TYPE_RESPONSE`	7	Response to a `RX_PACKET_TYPE_CHALLENGE` authentication challenge packet.
`RX_PACKET_TYPE_DEBUG`	8	Request for debugging information
`RX_N_PACKET_TYPES`	9	The number of *Rx* packet types defined above. Note that it also includes packet type 0 (which is unused) in the count

The `RX_PACKET_TYPES` definition provides a mapping of the above values to human-readable string names, and is exported by the `rx_packetTypes` variable catalogued in Section 5.4.9.

```
{"data",
  "ack",
  "busy",
  "abort",
  "ackall",
  "challenge",
  "response",
  "debug"
 }
```

5.2.12 Packet Classes

These definitions are used internally to manage alloction of *Rx* packet buffers according to quota classifications. Each packet belongs to one of the following classes, and its buffer is derived from the corresponding pool.

Programming Interface

Name	*Value*	*Description*
`RX_PACKET_CLASS_RECEIVE`	0	Receive packet for user data
`RX_PACKET_CLASS_SEND`	1	Send packet for user data
`RX_PACKET_CLASS_SPECIAL`	2	A special packet that does not holding user data, such as an acknowledgement or authentication challenge
`RX_N_PACKET_CLASSES`	3	The number of *Rx* packet classes defined above

5.2.13 Conditions Prompting Ack Packets

Rx acknowledgement packets are constructed and sent by the protocol according to the following reasons. These values appear in the *Rx* packet header of the ack packet itself.

Name	*Value*	*Description*
`RX_ACK_REQUESTED`	1	The peer has explicitly requested an ack on this packet
`RX_ACK_DUPLICATE`	2	A duplicate packet has been received
`RX_ACK_OUT_OF_SEQUENCE`	3	A packet has arrived out of sequence
`RX_ACK_EXCEEDS_WINDOW`	4	A packet sequence number higher than maximum value allowed by the call's window has been received
`RX_ACK_NOSPACE`	5	No packet buffer space is available
`RX_ACK_PING`	6	Acknowledgement for keep-alive purposes
`RX_ACK_PING_RESPONSE`	7	Response to a `RX_ACK_PING` packet
`RX_ACK_DELAY`	8	An ack generated due to a period of inactivity after normal packet receptions

5.2.14 Acknowledgement Types

These are the set of values placed into the `acks` array in an *Rx* acknowledgement packet, whose data format is defined by `struct rx_ackPacket`. These definitions are used to convey positive or negative acknowledgements for a given range of packets.

Name	*Value*	*Description*
`RX_ACK_TYPE_NACK`	0	Receiver doesn't currently have the associated packet; it may never have been received, or received and then later dropped before processing
`RX_ACK_TYPE_ACK`	1	Receiver has the associated packet queued, although it may later decide to discard it

5.2.15 Error Codes

Rx employs error codes ranging from -1 to -64. The *Rxgen* stub generator may use other error codes less than -64. User programs calling on *Rx*, on the other hand, are expected to return positive error codes. A return value of zero is interpreted as an indication that the given operation completed successfully.

Name	*Value*	*Description*
`RX_CALL_DEAD`	-1	A connection has been inactive past *Rx*'s tolerance levels and has been shut down.
`RX_INVALID_OPERATION`	-2	An invalid operation has been attempted, including such protocol errors as having a client-side call send data after having received the beginning of a reply from its server-side peer
`RX_CALL_TIMEOUT`	-3	The (optional) timeout value placed on this call has been exceeded (see Sections 5.5.3.4 and 5.6.5).
`RX_EOF`	-4	Unexpected end of data on a read operation
`RX_PROTOCOL_ERROR`	-5	An unspecified low-level *Rx* protocol error has occurred
`RX_USER_ABORT`	-6	A generic user abort code, used when no more specific error code needs to be communicated. For example, *Rx* clients employing the multicast feature (see Section 1.2.8) take advantage of this error code
`RX_ADDRINUSE`	-7	The given UDP port already in use (See the description of the *rx_Init()* function.)
`RX_DEBUGI_BADTYPE`	-8	Invalid debugging packet type was received

5.2.16 Debugging Values

Rx provides a set of data collections that convey information about its internal status and performance. The following values have been defined in support of this debugging and statistics-collection feature.

5.2.16.1 Version Information

Various versions of the *Rx* debugging/statistics interface are in existance, each defining different data collections and handling certain bugs. Each *Rx* facility is stamped with a version number of its debugging/statistics interface, allowing its clients to tailor their requests to the precise data collections that are supported by a particular *Rx* entity, and to properly interpret the data formats received through this interface. All existing *Rx* implementations should be at revision `M`.

Name	*Value*	*Description*
`RX_DEBUGI_VERSION_MINIMUM`	'L'	The earliest version of *Rx* statistics available
`RX_DEBUGI_VERSION`	'M'	The latest version of *Rx* statistics available
`RX_DEBUGI_VERSION_W_SECSTATS`	'L'	Identifies the earliest version in which statistics concerning *Rx* security objects is available
`RX_DEBUGI_VERSION_W_GETALLCONN`	'M'	The first version that supports getting information about all current *Rx* connections, as specified by the `RX_DEBUGI_GETALLCONN` debugging request packet opcode described below.
`RX_DEBUGI_VERSION_W_RXSTATS`	'M'	The first version that supports getting all the *Rx* statistics in one operation, as specified by the `RX_DEBUGI_RXSTATS` debugging request packet opcode described below.
`RX_DEBUGI_VERSION_W_UNALIGNED_CONN`	'L'	There was an alignment problem discovered when returning *Rx* connection information in older versions of this debugging/statistics interface. This identifies the last version that exhibited this alignment problem.

5.2.16.2 Opcodes

When requesting debugging/statistics information, the caller specifies one of the following supported data collections:

Name	*Value*	*Description*
RX_DEBUGI_GETSTATS	1	Get basic *Rx* statistics
RX_DEBUGI_GETCONN	2	Get information on all *Rx* connections considered "interesting" (as defined below), and no others
RX_DEBUGI_GETALLCONN	3	Get information on all existing *Rx* connection structures, even "uninteresting" ones
RX_DEBUGI_RXSTATS	4	Get all available *Rx* stats

An *Rx* connection is considered "interesting" if it is waiting for a call channel to free up or if it has been marked for destruction. If neither is true, a connection is still considered interesting if any of its call channels is actively handling a call or in its preparatory pre-call state. Failing all the above conditions, a connection is still tagged as interesting if any of its call channels is in either of the `RX_MODE_SENDING` or `RX_MODE_RECEIVING` modes, which are not allowed when the call is not active.

5.2.16.3 Queuing

These two queueing-related values indicate whether packets are present on the incoming and outgoing packet queues for a given *Rx* call. These values are *only* used in support of debugging and statistics-gathering operations.

Name	*Value*	*Description*
RX_OTHER_IN	1	Packets available in in queue
RX_OTHER_OUT	2	Packets available in out queue

5.3 Structures

This section describes the major exported *Rx* data structures of interest to application programmers. The following categories are utilized for the purpose of organizing the structure descriptions:

- Security objects
- Protocol objects
- Packet formats
- Debugging and statistics

- Miscellaneous

Please note that many fields described in this section are declared to be `VOID`. This is defined to be `char`, and is used to get around some compiler limitations.

5.3.1 Security Objects

As explained in Section 1.2.1, *Rx* provides a modular, extensible security model. This allows *Rx* applications to either use one of the built-in security/authentication protocol packages or write and plug in one of their own. This section examines the various structural components used by *Rx* to support generic security and authentication modules.

5.3.1.1 struct rx_securityOps

As previously described, each *Rx* security object must export a fixed set of interface functions, providing the full set of operations defined on the object. The `rx_securityOps` structure defines the array of functions comprising this interface. The *Rx* facility calls these routines at the appropriate times, without knowing the specifics of how any particular security object implements the operation.

A complete description of these interface functions, including information regarding their exact purpose, parameters, and calling conventions, may be found in Section 5.5.7.

Fields

int *(*op_Close)()* - React to the disposal of a security object.

int *(*op_NewConnection)()* - Invoked each time a new *Rx* connection utilizing the associated security object is created.

int *(*op_PreparePacket)()* - Invoked each time an outgoing *Rx* packet is created and sent on a connection using the given security object.

int *(*op_SendPacket)()* - Called each time a packet belonging to a call in a connection using the security object is physically transmitted.

int *(*op_CheckAuthentication)()* - This function is executed each time it is necessary to check whether authenticated calls are being perfomed on a connection using the associated security object.

int *(*op_CreateChallenge)()* - Invoked each time a server-side challenge event is created by *Rx*, namely when the identity of the principal associated with the peer process must be determined.

int *(*op_GetChallenge)()* - Called each time a client-side packet is constructed in response to an authentication challenge.

int *(*op_GetResponse)()* - Executed each time a response to a challenge event must be received on the server side of a connection.

int *(*op_CheckResponse)()* - Invoked each time a response to an authentication has been received, validating the response and pulling out the required authentication information.

int *(*op_CheckPacket) ()* - Invoked each time an *Rx* packet has been received, making sure that the packet is properly formatted and that it hasn't been altered.

int *(*op_DestroyConnection)()* - Called each time an *Rx* connection employing the given security object is destroyed.

int *(*op_GetStats)()* - Executed each time a request for statistics on the given security object has been received.

int *(*op_Spare1)()* **- int** *(*op_Spare3)()* - Three spare function slots, reserved for future use.

5.3.1.2 `struct rx_securityClass`

Variables of type `struct rx_securityClass` are used to represent instantiations of a particular security model employed by *Rx*. It consists of a pointer to the set of interface operations implementing the given security object, along with a pointer to private storage as necessary to support its operations. These security objects are also reference-counted, tracking the number of *Rx* connections in existance that use the given security object. If the reference count drops to zero, the security module may garbage-collect the space taken by the unused security object.

Fields

struct rx_securityOps *ops - Pointer to the array of interface functions for the security object.

VOID *privateData - Pointer to a region of storage used by the security object to support its operations.

int refCount - A reference count on the security object, tracking the number of *Rx* connections employing this model.

5.3.1.3 `struct rx_securityObjectStats`

This structure is used to report characteristics for an instantiation of a security object on a particular *Rx* connection, as well as performance figures for that object. It is used by the debugging portions of the *Rx* package. Every security object defines and manages fields such as `level` and `flags` differently.

Fields

char type - The type of security object being implemented. Existing values are:

- **0**: The null security package.
- **1**: An obsolete Kerberos-like security object.
- **2**: The rxkad discipline (see Chapter 3).

char level - The level at which encryption is utilized.

char sparec[10] - Used solely for alignment purposes.

long flags - Status flags regarding aspects of the connection relating to the security object.

u_long expires - Absolute time when the authentication information cached by the given connection expires. A value of zero indicates that the associated authentication information is valid for all time.

u_long packetsReceived - Number of packets received on this particular connection, and thus the number of incoming packets handled by the associated security object.

u_long packetsSent - Number of packets sent on this particular connection, and thus the number of outgoing packets handled by the associated security object.

u_long bytesReceived - Overall number of "payload" bytes received (i.e., packet bytes not associated with IP headers, UDP headers, and the security module's own header and trailer regions) on this connection.

u_long bytesSent - Overall number of "payload" bytes sent (i.e., packet bytes not associated with IP headers, UDP headers, and the security module's own header and trailer regions) on this connection.

short spares[4] - Several shortword spares, reserved for future use.

long sparel[8] - Several longword spares, reserved for future use.

5.3.2 Protocol Objects

The structures describing the main abstractions and entities provided by *Rx*, namely *services*, *peers*, *connections* and *calls* are covered in this section.

5.3.2.1 `struct rx_service`

An *Rx*-based server exports *services*, or specific RPC interfaces that accomplish certain tasks. Services are identified by *(host-address, UDP-port, serviceID)* triples. An *Rx* service is installed and initialized on a given host through the use of the *rx_NewService()* routine (See Section 5.6.3). Incoming calls are stamped with the *Rx* service type, and must match an installed service to be accepted. Internally, *Rx* services also carry string names for purposes of identification. These strings are useful to remote debugging and statistics-gathering programs. The use of a service ID allows a single server process to export multiple, independently-specified *Rx* RPC services.

Each *Rx* service contains one or more *security classes*, as implemented by individual *security objects*. These security objects implement end-to-end security protocols. Individual peer-to-peer *connections* established on behalf of an *Rx* service will select exactly one of the supported security objects to define the authentication procedures followed by all calls associated with the connection. Applications are not limited to using only the core set of built-in security objects offered by *Rx*. They are free to define their own security objects in order to execute the specific protocols they require.

It is possible to specify both the minimum and maximum number of lightweight processes available to handle simultaneous calls directed to an *Rx* service. In addition, certain procedures may be registered with the service and called at set times in the course of handling an RPC request.

Fields

u_short serviceId - The associated service number.

u_short servicePort - The chosen UDP port for this service.

char *serviceName - The human-readable service name, expressed as a character string.

osi_socket socket - The socket structure or file descriptor used by this service.

u_short nSecurityObjects - The number of entries in the array of supported security objects.

struct rx_securityClass **securityObjects - The array of pointers to the service's security class objects.

long (*executeRequestProc)() - A pointer to the routine to call when an RPC request is received for this service.

VOID (*destroyConnProc)() - A pointer to the routine to call when one of the server-side connections associated with this service is destroyed.

VOID (*newConnProc)() - A pointer to the routine to call when a server-side connection associated with this service is created.

VOID (*beforeProc)() - A pointer to the routine to call before an individual RPC call on one of this service's connections is executed.

VOID (*afterProc)() - A pointer to the routine to call after an individual RPC call on one of this service's connections is executed.

short nRequestsRunning - The number of simultaneous RPC calls currently in progress for this service.

short maxProcs - This field has two meanings. First, `maxProcs` limits the total number of requests that may execute in parallel for any one service. It also guarantees that this many requests may be handled in parallel if there are no active calls for any other service.

short minProcs - The minimum number of lightweight threads (hence requests) guaranteed to be simultaneously executable.

short connDeadTime - The number of seconds until a client of this service will be declared to be dead, if it is not responding to the RPC protocol.

short idleDeadTime - The number of seconds a server-side connection for this service will wait for packet I/O to resume after a quiescent period before the connection is marked as dead.

5.3.2.2 `struct rx_connection`

An *Rx* connection represents an authenticated communication path, allowing multiple asynchronous conversations (*calls*). Each connection is identified by a connection ID. The low-order bits of the connection ID are reserved so they may be stamped with the index of a particular call channel. With up to `RX_MAXCALLS` concurrent calls (set to 4 in this implementation), the bottom two bits are set aside for this purpose. The connection ID is not sufficient by itself to uniquely identify an *Rx* connection. Should a client crash and restart, it may reuse a connection ID, causing inconsistent results. In addition to the connection ID, the **epoch**, or start time for the client side of the connection, is used to identify a connection. Should the above scenario occur, a different epoch value will be chosen by the client, differentiating this incarnation from the orphaned connection record on the server side.

Each connection is associated with a parent service, which defines a set of supported security models. At creation time, an *Rx* connection selects the particular security protocol it will implement, referencing the associated service. The connection structure maintains state about the individual calls being simultaneously handled.

Fields

struct rx_connection *next - Used for internal queueing.

struct rx_peer *peer - Pointer to the connection's peer information (see below).

u_long epoch - Process start time of the client side of the connection.

u_long cid - Connection identifier. The call channel (i.e., the index into the connection's array of call structures) may appear in the bottom bits.

VOID *rock - Pointer to an arbitrary region of memory in support of the connection's operation. The contents of this area are opaque to the *Rx* facility in general, but are understood by any special routines used by this connection.

struct rx_call *call[RX_MAXCALLS] - Pointer to the call channel structures, describing up to `RX_MAXCALLS` concurrent calls on this connection.

u_long callNumber[RX_MAXCALLS] - The set of current call numbers on each of the call channels.

int timeout - Obsolete; no longer used.

u_char flags - Various states of the connection; see Section 5.2.4 for individual bit definitions.

u_char type - Whether the connection is a server-side or client-side one. See Section 5.2.5 for individual bit definitions.

u_short serviceId - The service ID that should be stamped on requests. This field is only used by client-side instances of connection structures.

struct rx_service *service - A pointer to the service structure associated with this connection. This field is only used by server-side instances of connection structures.

u_long serial - Serial number of the next outgoing packet associated with this connection.

u_long lastSerial - Serial number of the last packet received in association with this connection. This field is used in computing packet skew.

u_short secondsUntilDead - Maximum numer of seconds of silence that should be tolerated from the connection's peer before calls will be terminated with an `RX_CALL_DEAD` error.

u_char secondsUntilPing - The number of seconds between “pings” (keep-alive probes) when at least one call is active on this connection.

u_char securityIndex - The index of the security object being used by this connection. This number selects a slot in the security class array maintained by the service associated with the connection.

long error - Records the latest error code for calls occurring on this connection.

struct rx_securityClass *securityObject - A pointer to the security object used by this connection. This should coincide with the slot value chosen by the `securityIndex` field described above.

VOID *securityData - A pointer to a region dedicated to hosting any storage required by the security object being used by this connection.

u_short securityHeaderSize - The length in bytes of the portion of the packet header before the user's data that contains the security module's information.

u_short securityMaxTrailerSize - The length in bytes of the packet trailer, appearing after the user's data, as mandated by the connection's security module.

struct rxevent *challengeEvent - Pointer to an event that is scheduled when the server side of the connection is challenging the client to authenticate itself.

int lastSendTime - The last time a packet was sent on this connection.

long maxSerial - The largest serial number seen on incoming packets.

u_short hardDeadTime - The maximum number of seconds that any call on this connection may execute. This serves to throttle runaway calls.

5.3.2.3 `struct rx_peer`

For each connection, *Rx* maintains information describing the entity, or **peer**, on the other side of the wire. A peer is identified by a *(host, UDP-port)* pair. Included in the information kept on this remote communication endpoint are such network parameters as the maximum packet size supported by the host, current readings on round trip time to retransmission delays, and **packet skew** (see Section 1.2.7). There are also congestion control fields, ranging from descriptions of the maximum number of packets that may be sent to the peer without pausing and retransmission statistics. Peer structures are shared between connections whenever possible, and hence are reference-counted. A peer object may be garbage-collected if it is not actively referenced by any connection structure and a sufficient period of time has lapsed since the reference count dropped to zero.

Fields

struct rx_peer *next - Use to access internal lists.

u_long host - Remote IP address, in network byte order

u_short port - Remote UDP port, in network byte order

short packetSize - Maximum packet size for this host, if known.

u_long idleWhen - When the `refCount` reference count field (see below) went to zero.

short refCount - Reference count for this structure

u_char burstSize - Reinitialization size for the burst field (below).

u_char burst - Number of packets that can be transmitted immediately without pausing.

struct clock burstWait - Time delay until new burst aimed at this peer is allowed.

struct queue congestionQueue - Queue of RPC call descriptors that are waiting for a non-zero `burst` value.

int rtt - Round trip time to the peer, measured in milliseconds.

struct clock timeout - Current retransmission delay to the peer.

int nSent - Total number of distinct data packets sent, *not* including retransmissions.

int reSends - Total number of retransmissions for this peer since the peer structure instance was created.

u_long inPacketSkew - Maximum skew on incoming packets (see Section 1.2.7)

u_long outPacketSkew - Peer-reported maximum skew on outgoing packets (see Section 1.2.7).

5.3.2.4 `struct rx_call`

This structure records the state of an active call proceeding on a given *Rx* connection. As described above, each connection may have up to `RX_MAXCALLS` calls active at any one instant, and thus each connection maintains an array of `RX_MAXCALLS` `rx_call` structures. The information contained here is specific to the given call; "permanent" call state, such as the call number, is maintained in the connection structure itself.

Programming Interface

Fields

struct queue queue_item_header - Queueing information for this structure.

struct queue tq - Queue of outgoing ("transmit") packets.

struct queue rq - Queue of incoming ("receive") packets.

char *bufPtr - Pointer to the next byte to fill or read in the call's current packet, depending on whether it is being transmitted or received.

u_short nLeft - Number of bytes left to read in the first packet in the reception queue (see field `rq`).

u_short nFree - Number of bytes still free in the last packet in the transmission queue (see field `tq`).

struct rx_packet *currentPacket - Pointer to the current packet being assembled or read.

struct rx_connection *conn - Pointer to the parent connection for this call.

u_long *callNumber - Pointer to call number field within the call's current packet.

u_char channel - Index within the parent connection's call array that describes this call.

u_char dummy1, dummy2 - These are spare fields, reserved for future use.

u_char state - Current call state. The associated bit definitions appear in Section 5.2.7.

u_char mode - Current mode of a call that is in `RX_STATE_ACTIVE` state. The associated bit definitions appear in Section 5.2.8.

u_char flags - Flags pertaining to the state of the given call. The associated bit definitions appear in Section 5.2.7.

u_char localStatus - Local user status information, sent out of band. This field is currently not in use, set to zero.

u_char remoteStatus - Remote user status information, received out of band. This field is currently not in use, set to zero.

long error - Error condition for this call.

u_long timeout - High level timeout for this call

u_long rnext - Next packet sequence number expected to be received.

u_long rprev - Sequence number of the previous packet received. This number is used to decide the proper sequence number for the next packet to arrive, and may be used to generate a negative acknowledgement.

u_long rwind - Width of the packet receive window for this call. The peer must not send packets with sequence numbers greater than or equal to `rnext + rwind`.

u_long tfirst - Sequence number of the first unacknowledged transmit packet for this call.

u_long tnext - Next sequence number to use for an outgoing packet.

u_long twind - Width of the packet transmit window for this call. *Rx* cannot assign a sequence number to an outgoing packet greater than or equal to `tfirst + twind`.

struct rxevent *resendEvent - Pointer to a pending retransmission event, if any.

struct rxevent *timeoutEvent - Pointer to a pending timeout event, if any.

struct rxevent *keepAliveEvent - Pointer to a pending keep-alive event, if this is an active call.

struct rxevent *delayedAckEvent - Pointer to a pending delayed acknowledgement packet event, if any. Transmission of a delayed acknowledgement packet is scheduled after all outgoing packets for a call have been sent. If neither a reply nor a new call are received by the time the `delayedAckEvent` activates, the ack packet will be sent.

int lastSendTime - Last time a packet was sent for this call.

int lastReceiveTime - Last time a packet was received for this call.

VOID (*arrivalProc)() - Pointer to the procedure to call when reply is received.

VOID *arrivalProcHandle - Pointer to the handle to pass to the `arrivalProc` as its first argument.

VOID *arrivalProcArg - Pointer to an additional argument to pass to the given `arrivalProc`.

u_long lastAcked - Sequence number of the last packet "hard-acked" by the receiver. A packet is considered to be hard-acked if an acknowledgement is generated *after* the reader has processed it. The *Rx* facility may sometimes "soft-ack" a windowfull of packets before they have been picked up by the receiver.

u_long startTime - The time this call started running.

u_long startWait - The time that a server began waiting for input data or send quota.

5.3.3 Packet Formats

The following sections cover the different data formats employed by the suite of *Rx* packet types, as enumerated in Section 5.2.11. A description of the most commonly-employed *Rx* packet header appears first, immediately followed by a description of the generic packet container and descriptor. The formats for *Rx* acknowledgement packets and debugging/statistics packets are also examined.

5.3.3.1 `struct rx_header`

Every *Rx* packet has its own header region, physically located after the leading IP/UDP headers. This header contains connection, call, security, and sequencing information. Along with a type identifier, these fields allow the receiver to properly interpret the packet. In addition, every client relates its "epoch", or *Rx* incarnation date, in each packet. This assists in identifying protocol problems arising from reuse of connection identifiers due to a client restart. Also included in the header is a byte of user-defined status information, allowing out-of-band channel of communication for the higher-level application using *Rx* as a transport mechanism.

Fields

u_long epoch - Birth time of the client *Rx* facility.

u_long cid - Connection identifier, as defined by the client. The last `RX_CIDSHIFT` bits in the `cid` field identify which of the server-side `RX_MAXCALLS` call channels is to receive the packet.

u_long callNumber - The current call number on the chosen call channel.

u_long seq - Sequence number of this packet. Sequence numbers start with 0 for each new *Rx* call.

u_long serial - This packet's serial number. A new serial number is stamped on each packet transmitted (or retransmitted).

u_char type - What type of *Rx* packet this is; see Section 5.2.11 for the list of legal definitions.

u_char flags - Flags describing this packet; see Section 5.2.9 for the list of legal settings.

u_char userStatus - User-defined status information, uninterpreted by the *Rx* facility itself. This field may be easily set or retrieved from *Rx* packets via calls to the *rx_GetLocalStatus()*, *rx_SetLocalStatus()*, *rx_GetRemoteStatus()*, and *rx_SetRemoteStatus()* macros.

u_char securityIndex - Index in the associated server-side service class of the security object used by this call.

u_short serviceId - The server-provided service ID to which this packet is directed.

u_short spare - This field was originally a true spare, but is now used by the built-in **rxkad** security module for packet header checksums. See the descriptions of the related *rx_IsUsingPktChecksum()*, *rx_GetPacketCksum()*, and *rx_SetPacketCksum()* macros.

5.3.3.2 `struct rx_packet`

This structure is used to describe an *Rx* packet, and includes the wire version of the packet contents, where all fields exist in network byte order. It also includes acknowledgement, length, type, and queueing information.

Fields

struct queue queueItemHeader - Field used for internal queueing.

u_char acked - If non-zero, this field indicates that this packet has been *tentatively* (soft-) acknowledged. Thus, the packet has been accepted by the *rx* peer entity on the other side of the connection, but has not yet necessarily been passed to the true reader. The sender is not free to throw the packet away, as it might still get dropped by the peer before it is delivered to its destination process.

short length - Length in bytes of the user data section.

u_char packetType - The type of *Rx* packet described by this record. The set of legal choices is available in Section 5.2.11.

struct clock retryTime - The time when this packet should be retransmitted next.

struct clock timeSent - The last time this packet was transmitted.

struct rx_header header - A copy of the internal *Rx* packet header.

wire - The text of the packet as it appears on the wire. This structure has the following sub-fields:

- **u_long head[RX_HEADER_SIZE/sizeof(long)]** The wire-level contents of IP, UDP, and *Rx* headers.
- **u_long data[RX_MAX_PACKET_DATA_SIZE/sizeof(long)]** The wire form of the packet's "payload", namely the user data it carries.

5.3.3.3 `struct rx_ackPacket`

This is the format for the data portion of an *Rx* acknowledgement packet, used to inform a peer entity performing packet transmissions that a subset of its packets has been properly received.

Fields

u_short bufferSpace - Number of packet buffers available. Specifically, the number of packet buffers that the ack packet's sender is willing to provide for data on this or subsequent calls. This number does not have to fully accurate; it is acceptable for the sender to provide an estimate.

u_short maxSkew - The maximum difference seen between the serial number of the packet being acknowledged and highest packet yet received. This is an indication of the degree to which packets are arriving out of order at the receiver.

u_long firstPacket - The serial number of the first packet in the list of acknowledged packets, as represented by the `acks` field below.

u_long previousPacket - The previous packet serial number received.

u_long serial - The serial number of the packet prompted the acknowledgement.

u_char reason - The reason given for the acknowledgement; legal values for this field are described in Section 5.2.13.

u_char nAcks - Number of acknowledgements active in the `acks` array immediately following.

u_char acks[RX_MAXACKS] - Up to `RX_MAXACKS` packet acknowledgements. The legal values for each slot in the `acks` array are described in Section 5.2.14. Basically, these fields indicate either positive or negative acknowledgements.

All packets with serial numbers prior to `FirstPacket` are implicitly acknowledged by this packet, indicating that they have been fully processed by the receiver. Thus, the sender need no longer be concerned about them, and may release all of the resources that they occupy. Packets with serial numbers `firstPacket + nAcks` and higher are not acknowledged by this ack packet. Packets with serial numbers in the range [`firstPacket, firstPacket + nAcks`) are explicitly acknowledged, yet their sender-side resources must not yet be released, as there is yet no guarantee that the receiver will not throw them away before they can be processed there.

There are some details of importance to be noted. For one, receiving a positive acknowlegement via the `acks` array does *not* imply that the associated packet is immune from being dropped before it is read and processed by the receiving entity. It does, however, imply that the sender should stop retransmitting the packet until further notice. Also, arrival of an ack packet should prompt the transmitter to immediately retransmit all packets it holds that have not been explicitly acknowledged and that were last transmitted with a serial number less than the highest serial number acknowledged by the `acks` array.

Note: The fields in this structure are always kept in wire format, namely in network byte order.

5.3.4 Debugging and Statistics

The following structures are defined in support of the debugging and statistics-gathering interfaces provided by *Rx*.

5.3.4.1 `struct rx_stats`

This structure maintains *Rx* statistics, and is gathered by such tools as the `rxdebug` program. It must be possible for all of the fields placed in this structure to be successfully converted from their on-wire network byte orderings to the host-specific ordering.

Fields

- **int packetRequests** - Number of packet allocation requests processed.
- **int noPackets[RX_N_PACKET_CLASSES]** - Number of failed packet requests, organized per allocation class.
- **int socketGreedy** - Whether the `SO_GREEDY` setting succeeded for the *Rx* socket.
- **int bogusPacketOnRead** - Number of inappropriately short packets received.
- **int bogusHost** - Contains the host address from the last bogus packet received.
- **int noPacketOnRead** - Number of attempts to read a packet off the wire when there was actually no packet there.
- **int noPacketBuffersOnRead** - Number of dropped data packets due to lack of packet buffers.
- **int selects** - Number of selects waiting for a packet arrival or a timeout.
- **int sendSelects** - Number of selects forced when sending packets.
- **int packetsRead[RX_N_PACKET_TYPES]** - Total number of packets read, classified by type.

int dataPacketsRead - Number of unique data packets read off the wire.

int ackPacketsRead - Number of ack packets read.

int dupPacketsRead - Number of duplicate data packets read.

int spuriousPacketsRead - Number of inappropriate data packets.

int packetsSent[RX_N_PACKET_TYPES] - Number of packet transmissions, broken down by packet type.

int ackPacketsSent - Number of ack packets sent.

int pingPacketsSent - Number of ping packets sent.

int abortPacketsSent - Number of abort packets sent.

int busyPacketsSent - Number of busy packets sent.

int dataPacketsSent - Number of unique data packets sent.

int dataPacketsReSent - Number of retransmissions.

int dataPacketsPushed - Number of retransmissions pushed early by a negative acknowledgement.

int ignoreAckedPacket - Number of packets not retransmitted because they have already been acked.

int struct clock totalRtt - Total round trip time measured for packets, used to compute average time figure.

struct clock minRtt - Minimum round trip time measured for packets.

struct clock maxRtt - Maximum round trip time measured for packets.

int nRttSamples - Number of round trip samples.

int nServerConns - Number of server connections.

int nClientConns - Number of client connections.

int nPeerStructs - Number of peer structures.

int nCallStructs - Number of call structures physically allocated (using the internal storage allocator routine).

int nFreeCallStructs - Number of call structures which were pulled from the free queue, thus avoiding a call to the internal storage allocator routine.

int spares[10] - Ten integer spare fields, reserved for future use.

5.3.4.2 `struct rx_debugIn`

This structure defines the data format for a packet requesting one of the statistics collections maintained by *Rx*.

Fields

- **long type** - The specific data collection that the caller desires. Legal settings for this field are described in Section 5.2.16.2.
- **long index** - This field is only used when gathering information on *Rx* connections. Choose the index of the server-side connection record of which we are inquiring. This field may be used as an iterator, stepping through all the connection records, one per debugging request, until they have all been examined.

5.3.4.3 `struct rx_debugStats`

This structure describes the data format for a reply to an `RX_DEBUGI_GETSTATS` debugging request packet. These fields are given values indicating the current state of the *Rx* facility.

Fields

- **long nFreePackets** - Number of packet buffers currently assigned to the free pool.
- **long packetReclaims** - *Currently unused.*
- **long callsExecuted** - Number of calls executed since the *Rx* facility was initialized.
- **char waitingForPackets** - Is *Rx* currently blocked waiting for a packet buffer to come free?
- **char usedFDs** - If the *Rx* facility is executing in the kernel, return the number of UNIX file descriptors in use. This number is not directly related to the *Rx* package, but rather describes the state of the machine on which *Rx* is running.
- **char version** - Version number of the debugging package.
- **char spare1[1]** - Byte spare, reserved for future use.
- **long spare2[10]** - Set of 10 longword spares, reserved for future use.

5.3.4.4 `struct rx_debugConn`

This structure defines the data format returned when a caller requests information concerning an *Rx* connection. Thus, `rx_debugConn` defines the external packaging of interest to external parties. Most of these fields are set from the `rx_connection` structure, as defined in Section 5.3.2.2, and others are obtained by indirecting through such objects as the connection's peer and call structures.

Fields

long host - Address of the host identified by the connection's peer structure.

long cid - The connection ID.

long serial - The serial number of the next outgoing packet associated with this connection.

long callNumber[RX_MAXCALLS] - The current call numbers for the individual call channels on this connection.

long error - Records the latest error code for calls occurring on this connection.

short port - UDP port associated with the connection's peer.

char flags - State of the connection; see Section 5.2.4 for individual bit definitions.

char type - Whether the connection is a server-side or client-side one. See Section 5.2.5 for individual bit definitions.

char securityIndex - Index in the associated server-side service class of the security object being used by this call.

char sparec[3] - Used to force alignment for later fields.

char callState[RX_MAXCALLS] - Current call state on each call channel. The associated bit definitions appear in Section 5.2.7.

char callMode[RX_MAXCALLS] - Current mode of all call channels that are in `RX_STATE_ACTIVE` state. The associated bit definitions appear in Section 5.2.8.

char callFlags[RX_MAXCALLS] - Flags pertaining to the state of each of the connection's call channels. The associated bit definitions appear in Section 5.2.7.

char callOther[RX_MAXCALLS] - Flag field for each call channel, where the presence of the `RX_OTHER_IN` flag indicates that there are packets present on the given call's reception queue, and the `RX_OTHER_OUT` flag indicates the presence of packets on the transmission queue.

struct rx_securityObjectStats secStats - The contents of the statistics related to the security object selected by the `securityIndex` field, if any.

long epoch - The connection's client-side incarnation time.

long sparel[10] - A set of 10 longword fields, reserved for future use.

5.3.4.5 struct rx_debugConn_vL

This structure is identical to `rx_debugConn` defined above, except for the fact that it is missing the `sparec` field. This `sparec` field is used in `rx_debugConn` to fix an alignment problem that was discovered in version `L` of the debugging/statistics interface (hence the trailing "tt _vL tag in the structure name). This alignment problem is fixed in version `M`, which utilizes and exports the rx_debugConn structure exclusively. Information regarding the range of version-numbering values for the *Rx* debugging/statistics interface may be found in Section 5.2.16.1.

5.4 Exported Variables

This section describes the set of variables that the *Rx* facility exports to its applications. Some of these variables have macros defined for the sole purpose of providing the caller with a convenient way to manipulate them. Note that some of these exported variables are never meant to be altered by application code (e.g., `rx_nPackets`).

5.4.1 rx_connDeadTime

This integer-valued variable determines the maximum number of seconds that a connection may remain completely inactive, without receiving packets of any kind, before it is eligible for garbage collection. Its initial value is 12 seconds. The *rx_SetRxDeadTime* macro sets the value of this variable.

5.4.2 rx_idleConnectionTime

This integer-valued variable determines the maximum number of seconds that a server connection may "idle" (i.e., not have any active calls and otherwise not have sent a packet) before becoming eligible for garbage collection. Its initial value is 60 seconds.

5.4.3 rx_idlePeerTime

This integer-valued variable determines the maximum number of seconds that an *Rx* peer structure is allowed to exist without any connection structures referencing it before becoming eligible for garbage collection. Its initial value is 60 seconds.

5.4.4 rx_extraQuota

This integer-valued variable is part of the *Rx* packet quota system (see Section 1.2.6), which is used to avoid system deadlock. This ensures that each server-side thread has a minimum number of packets at its disposal, allowing it to continue making progress on active calls. This particular variable records how many extra data packets a user has requested be allocated. Its initial value is 0.

5.4.5 rx_extraPackets

This integer-valued variable records how many additional packet buffers are to be created for each *Rx* server thread. The caller, upon setting this variable, is applying some application-specific knowledge of the level of network activity expected. The `rx_extraPackets` variable is used to compute the overall number of packet buffers to reserve per server thread, namely `rx_nPackets`, described below. The initial value is 32 packets.

5.4.6 rx_nPackets

This integer-valued variable records the total number of packet buffers to be allocated per *Rx* server thread. It takes into account the quota packet buffers and the extra buffers requested by the caller, if any.

Note: This variable should **never** be set directly; the *Rx* facility itself computes its value. Setting it incorrectly may result in the service becoming deadlocked due to insufficient resources. Callers wishing to allocate more packet buffers to their server threads should indicate that desire by setting the `rx_extraPackets` variable described above.

5.4.7 rx_nFreePackets

This integer-valued variable records the number of *Rx* packet buffers not currently used by any call. These unused buffers are collected into a free pool.

5.4.8 rx_stackSize

This integer-valued variable records the size in bytes for the lightweight process stack. The variable is initially set to `RX_DEFAULT_STACK_SIZE`, and is typically manipulated via the *rx_SetStackSize()* macro.

5.4.9 rx_packetTypes

This variable holds an array of string names used to describe the different roles for *Rx* packets. Its value is derived from the `RX_PACKET_TYPES` definition found in Section 5.2.11.

5.4.10 rx_stats

This variable contains the statistics structure that keeps track of *Rx* statistics. The `struct rx_stats` structure it provides is defined in Section 5.3.4.1.

5.5 Macros

Rx uses many macro definitions in preference to calling C functions directly. There are two main reasons for doing this:

- **Field selection**: Many *Rx* operations are easily realized by returning the value of a particular structure's field. It is wasteful to invoke a C routine to simply fetch a structure's field, incurring unnecessary function call overhead. Yet, a convenient, procedure-oriented operation is still provided to *Rx* clients for such operations by the use of macros. For example, the *rx_ConnectionOf()* macro, described in Section 5.5.1.1, simply indirects through the *Rx* call structure pointer parameter to deliver the `conn` field.
- **Performance optimization**: In some cases, a simple test or operation can be performed to accomplish a particular task. When this simple, straightforward operation fails, then a true C routine may be called to handle to more complex (and rarer) situation. The *Rx* macro *rx_Write()*, described in Section 5.5.6.2, is a perfect example of this type of optimization. Invoking *rx_Write()* first checks to determine whether or not the outgoing call's internal buffer has enough room to accept the specified data bytes. If so, it copies them into the call's buffer, updating

counts and pointers as appropriate. Otherwise, *rx_Write()* calls the *rx_WriteProc()* to do the work, which in this more complicated case involves packet manipulations, dispatches, and allocations. The result is that the common, simple cases are often handled in-line, with more complex (and rarer) cases handled through true function invocations.

The set of *Rx* macros is described according to the following categories.

- Field selections/assignments
- Boolean operations
- Service attributes
- Security-related operations
- Sizing operations
- Complex operation
- Security operation invocations

5.5.1 Field Selections/Assignments

These macros facilitate the fetching and setting of fields from the structures described Chapter 5.3.

5.5.1.1 *rx_ConnectionOf()*

```
#define rx_ConnectionOf(call) ((call)->conn)
```

Generate a reference to the connection field within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. An application of the *rx_ConnectionOf()* macro itself yields an object of type `rx_peer`.

5.5.1.2 *rx_PeerOf()*

```
#define rx_PeerOf(conn) ((conn)->peer)
```

Generate a reference to the peer field within the given *Rx* call structure. The value supplied as the `conn` argument must resolve into an object of type `(struct rx_connection *)`. An instance of the *rx_PeerOf()* macro itself resolves into an object of type `rx_peer`.

5.5.1.3 *rx_HostOf()*

```
#define rx_HostOf(peer) ((peer)->host)
```

Generate a reference to the host field within the given *Rx* peer structure. The value supplied as the `peer` argument must resolve into an object of type `(struct rx_peer *)`. An instance of the *rx_HostOf()* macro itself resolves into an object of type `u_long`.

5.5.1.4 *rx_PortOf()*

```
#define rx_PortOf(peer) ((peer)->port)
```

Generate a reference to the port field within the given *Rx* peer structure. The value supplied as the `peer` argument must resolve into an object of type `(struct rx_peer *)`. An instance of the *rx_PortOf()* macro itself resolves into an object of type `u_short`.

5.5.1.5 *rx_GetLocalStatus()*

```
#define rx_GetLocalStatus(call, status) ((call)->localStatus)
```

Generate a reference to the `localStatus` field, which specifies the local user status sent out of band, within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. The second argument, `status`, is not used. An instance of the *rx_GetLocalStatus()* macro itself resolves into an object of type `u_char`.

5.5.1.6 *rx_SetLocalStatus()*

```
#define rx_SetLocalStatus(call, status)
            ((call)->localStatus = (status))
```

Assign the contents of the `localStatus` field, which specifies the local user status sent out of band, within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. The second argument, `status`, provides the new value of the `localStatus` field, and must resolve into an object of type `u_char`. An instance of the *rx_GetLocalStatus()* macro itself resolves into an object resulting from the assignment, namely the `u_char status` parameter.

5.5.1.7 *rx_GetRemoteStatus()*

```
#define rx_GetRemoteStatus(call) ((call)->remoteStatus)
```

Generate a reference to the `remoteStatus` field, which specifies the remote user status received out of band, within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. An instance of the *rx_GetRemoteStatus()* macro itself resolves into an object of type `u_char`.

5.5.1.8 *rx_Error()*

```
#define rx_Error(call) ((call)->error)
```

Generate a reference to the `error` field, which specifies the current error condition, within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. An instance of the *rx_Error()* macro itself resolves into an object of type `long`.

5.5.1.9 *rx_DataOf()*

```
#define rx_DataOf(packet) ((char *) (packet)->wire.data)
```

Generate a reference to the beginning of the data portion within the given *Rx* packet as it appears on the wire. Any encryption headers will be resident at this address. For *Rx* packets of type `RX_PACKET_TYPE_DATA`, the actual user data will appear at the address returned by the *rx_DataOf* macro plus the connection's security header size. The value supplied as the `packet` argument must resolve into an object of type `(struct rx_packet *)`. An instance of the *rx_DataOf()* macro itself resolves into an object of type `(u_long *)`.

5.5.1.10 *rx_GetDataSize()*

```
#define rx_GetDataSize(packet) ((packet)->length)
```

Generate a reference to the `length` field, which specifies the number of bytes of user data contained within the wire form of the packet, within the given *Rx* packet description structure. The value supplied as the `packet` argument must resolve into an object of type `(struct rx_packet *)`. An instance of the *rx_GetDataSize()* macro itself resolves into an object of type `short`.

5.5.1.11 *rx_SetDataSize()*

```
#define rx_SetDataSize(packet, size) ((packet)->length = (size))
```

Assign the contents of the `length` field, which specifies the number of bytes of user data contained within the wire form of the packet, within the given *Rx* packet description structure. The value supplied as the `packet` argument must resolve into an object of type `(struct rx_packet *)`. The second argument, `size`, provides the new value of the `length` field, and must resolve into an object of type `short`. An instance of the *rx_SetDataSize()* macro itself resolves into an object resulting from the assignment, namely the `short length` parameter.

5.5.1.12 *rx_GetPacketCksum()*

```
#define rx_GetPacketCksum(packet) ((packet)->header.spare)
```

Generate a reference to the header checksum field, as used by the built-in **rxkad** security module (See Chapter 3), within the given *Rx* packet description structure. The value supplied as the `packet` argument must resolve into an object of type `(struct rx_packet *)`. An instance of the *rx_GetPacketCksum()* macro itself resolves into an object of type `u_short`.

5.5.1.13 *rx_SetPacketCksum()*

```
#define rx_SetPacketCksum(packet, cksum)
            ((packet)->header.spare = (cksum))
```

Assign the contents of the header checksum field, as used by the built-in **rxkad** security module (See Chapter 3), within the given *Rx* packet description structure. The value supplied as the `packet` argument must resolve into an object of type `(struct rx_packet *)`. The second argument, `cksum`, provides the new value of the checksum, and must resolve into an object of type `u_short`. An instance of the *rx_SetPacketCksum()* macro itself resolves into an object resulting from the assignment, namely the `u_short` checksum parameter.

5.5.1.14 *rx_GetRock()*

```
#define rx_GetRock(obj, type) ((type)(obj)->rock)
```

Generate a reference to the field named `rock` within the object identified by the `obj` pointer. One common *Rx* structure to which this macro may be applied is `struct rx_connection`. The specified `rock` field is casted to the value of the `type` parameter, which is the overall value of the *rx_GetRock()* macro.

5.5.1.15 *rx_SetRock()*

```
#define rx_SetRock(obj, newrock) ((obj)->rock = (VOID *)(newrock))
```

Assign the contents of the `newrock` parameter into the `rock` field of the object pointed to by `obj`. The given object's `rock` field must be of type `(VOID *)`. An instance of the *rx_SetRock()* macro itself resolves into an object resulting from the assignment and is of type `(VOID *)`.

5.5.1.16 *rx_SecurityClassOf()*

```
#define rx_SecurityClassOf(conn) ((conn)->securityIndex)
```

Generate a reference to the security index field of the given *Rx* connection description structure. This identifies the security class used by the connection. The value supplied as the `conn` argument must resolve into an object of type `(struct rx_connection *)`. An instance of the *rx_SecurityClassOf()* macro itself resolves into an object of type `u_char`.

5.5.1.17 *rx_SecurityObjectOf()*

```
#define rx_SecurityObjectOf(conn) ((conn)->securityObject)
```

Generate a reference to the security object in use by the given *Rx* connection description structure. The choice of security object determines the authentication protocol enforced by the connection. The value supplied as the `conn` argument must resolve into an object of type `(struct rx_connection *)`. An instance of the *rx_SecurityObjectOf()* macro itself resolves into an object of type `(struct rx_securityClass *)`.

5.5.2 Boolean Operations

The macros described in this section all return Boolean values. They are used to query such things as the whether a connection is a server-side or client-side one and if extra levels of checksumming are being used in *Rx* packet headers.

5.5.2.1 *rx_IsServerConn()*

```
#define rx_IsServerConn(conn) ((conn)->type == RX_SERVER_CONNECTION)
```

Determine whether or not the *Rx* connection specified by the `conn` argument is a server-side connection. The value supplied for `conn` must resolve to an object of type `struct rx_connection`. The result is determined by testing whether or not the connection's `type` field is set to `RX_SERVER_CONNECTION`.

Note: Another macro, *rx_ServerConn()*, performs the identical operation.

5.5.2.2 *rx_IsClientConn()*

```
#define rx_IsClientConn(conn) ((conn)->type == RX_CLIENT_CONNECTION)
```

Determine whether or not the *Rx* connection specified by the `conn` argument is a client-side connection. The value supplied for `conn` must resolve to an object of type `struct rx_connection`. The result is determined by testing whether or not the connection's `type` field is set to `RX_CLIENT_CONNECTION`.

Note: Another macro, *rx_ClientConn()*, performs the identical operation.

5.5.2.3 *rx_IsUsingPktCksum()*

```
#define rx_IsUsingPktCksum(conn)
            ((conn)->flags & RX_CONN_USING_PACKET_CKSUM)
```

Determine whether or not the *Rx* connection specified by the `conn` argument is checksumming the headers of all packets on its calls. The value supplied for `conn` must resolve to an object of type `struct rx_connection`. The result is determined by testing whether or not the connection's `flags` field has the `RX_CONN_USING_PACKET_CKSUM` bit enabled.

5.5.3 Service Attributes

This section describes user-callable macros that manipulate the attributes of an *Rx* service. Note that these macros *must* be called (and hence their operations performed) before the given service is installed via the appropriate invocation of the associated *rx_StartServer()* function.

5.5.3.1 *rx_SetStackSize()*

```
#define rx_SetStackSize(service, stackSize)
    rx_stackSize = (((stackSize) > rx_stackSize) ?
                        stackSize : rx_stackSize)
```

Inform the *Rx* facility of the stack size in bytes for a class of threads to be created in support of *Rx* services. The exported `rx_stackSize` variable tracks the high-water mark for all stack size requests before the call to *rx_StartServer()*. If no calls to *rx_SetStackSize()* are made, then `rx_stackSize` will retain its default setting of `RX_DEFAULT_STACK_SIZE`.

In this macro, the first argument is not used. It was originally intended that thread stack sizes would be settable on a per-service basis. However, calls to *rx_SetStackSize()* will ignore the service parameter and set the high-water mark for all *Rx* threads created after the use of *rx_SetStackSize()*. The second argument, `stackSize`, specifies determines the new stack size, and should resolve to an object of type `int`. The value placed in the `stackSize` parameter will not be recorded in the global `rx_stackSize` variable unless it is greater than the variable's current setting.

An instance of the *rx_SetStackSize()* macro itself resolves into the result of the assignment, which is an object of type `int`.

5.5.3.2 *rx_SetMinProcs()*

```
#define rx_SetMinProcs(service, min) ((service)->minProcs = (min))
```

Choose `min` as the minimum number of threads guaranteed to be available for parallel execution of the given *Rx* `service`. The `service` parameter should resolve to an object of type `struct rx_service`. The `min` parameter should resolve to an object of type `short`. An instance of the *rx_SetMinProcs()* macro itself resolves into the result of the assignment, which is an object of type `short`.

5.5.3.3 *rx_SetMaxProcs()*

```
#define rx_SetMaxProcs(service, max) ((service)->maxProcs = (max))
```

Limit the maximum number of threads that may be made available to the given *Rx* `service` for parallel execution to be `max`. The `service` parameter should resolve to an object of type `struct rx_service`. The `max` parameter should resolve to an object of type `short`. An instance of the *rx_SetMaxProcs()* macro itself resolves into the result of the assignment, which is an object of type `short`.

5.5.3.4 *rx_SetIdleDeadTime()*

```
#define rx_SetIdleDeadTime(service, time)
            ((service)->idleDeadTime = (time))
```

Every *Rx* service has a maximum amount of time it is willing to have its active calls sit idle (i.e., no new data is read or written for a call marked as `RX_STATE_ACTIVE`) before unilaterally shutting down the call. The expired call will have its error field set to `RX_CALL_TIMEOUT`. The operative assumption in this situation is that the client code is exhibiting a protocol error that prevents progress from being made on this call, and thus the call's resources on the server side should be freed. The default value, as recorded in the service's `idleDeadTime` field, is set at service creation time to be 60 seconds. The *rx_SetIdleTime()* macro allows a caller to dynamically set this idle call timeout value.

The `service` parameter should resolve to an object of type `struct rx_service`. Also, the `time` parameter should resolve to an object of type `short`. Finally, an instance of the *rx_SetIdleDeadTime()* macro itself resolves into the result of the assignment, which is an object of type `short`.

5.5.3.5 *rx_SetServiceDeadTime()*

```
#define rx_SetServiceDeadTime(service, seconds)
            ((service)->secondsUntilDead = (seconds))
```

Note: This macro definition is obsolete and should NOT be used. Including it in application code will generate a compile-time error, since the service structure no longer has such a field defined.

See the description of the *rx_SetConnDeadTime()* macro below to see how hard timeouts may be set for situations of complete call inactivity.

5.5.3.6 *rx_SetRxDeadTime()*

```
#define rx_SetRxDeadTime(seconds) (rx_connDeadTime = (seconds))
```

Inform the *Rx* facility of the maximum number of seconds of complete inactivity that will be tolerated on an active call. The exported `rx_connDeadTime` variable tracks this value, and is initialized to a value of 12 seconds. The current value of *rx_connDeadTime* will be copied into new *Rx* service and connection records upon their creation.

The `seconds` argument determines the value of `rx_connDeadTime`, and should resolve to an object of type `int`. An instance of the *rx_SetRxDeadTime()* macro itself resolves into the result of the assignment, which is an object of type `int`.

5.5.3.7 *rx_SetConnDeadTime()*

```
#define rx_SetConnDeadTime(conn, seconds)
            (rxi_SetConnDeadTime(conn, seconds))
```

Every *Rx* connection has a maximum amount of time it is willing to have its active calls on a server connection sit without receiving packets of any kind from its peer. After such a quiescent time, during which neither data packets (regardless of whether they are properly sequenced or duplicates) nor keep-alive packets are received, the call's error field is set to `RX_CALL_DEAD` and the call is terminated. The operative assumption in this situation is that the client making the call has perished, and thus the call's resources on the server side should be freed. The default value, as recorded in the connection's `secondsUntilDead` field, is set at connection creation time to be the same as its parent service. The *rx_SetConnDeadTime()* macro allows a caller to dynamically set this timeout value.

The `conn` parameter should resolve to an object of type `struct rx_connection`. Also, the `seconds` parameter should resolve to an object of type `int`. Finally, an instance of the *rx_SetConnDeadTime()* macro itself resolves into the a call to *rxi_SetConnDeadTime()*, whose return value is `void`.

5.5.3.8 *rx_SetConnHardDeadTime()*

```
#define rx_SetConnHardDeadTime(conn, seconds)
            ((conn)->hardDeadTime = (seconds))
```

It is convenient to be able to specify that calls on certain *Rx* connections have a hard absolute timeout. This guards against protocol errors not caught by other checks in which one or both of the client and server are looping. The *rx_SetConnHardDeadTime()* macro is available for this purpose. It will limit calls on the connection identified by the `conn` parameter to execution times of no more than the given number of `seconds`. By default, active calls on an *Rx* connection may proceed for an unbounded time, as long as they are not totally quiescent (see Section 5.5.3.7 for a description of the *rx_SetConnDeadTime()*) or idle (see Section 5.5.3.4 for a description of the *rx_SetIdleDeadTime()*).

The `conn` parameter should resolve to an object of type `(struct rx_connection *)`. The `seconds` parameter should resolve to an object of type `u_short`. An instance of the *rx_SetConnHardDeadTime()* macro itself resolves into the result of the assignment, which is an object of type `u_short`.

5.5.3.9 *rx_GetBeforeProc()*

```
#define rx_GetBeforeProc(service) ((service)->beforeProc)
```

Return a pointer of type `(VOID *)()` to the procedure associated with the given *Rx* `service` that will be called immediately upon activation of a server thread to handle an incoming call. The `service` parameter should resolve to an object of type `struct rx_service`.

When an *Rx* service is first created (via a call to the *rx_NewService()* function), its `beforeProc` field is set to a null pointer. See the description of the *rx_SetBeforeProc()* below.

5.5.3.10 *rx_SetBeforeProc()*

```
#define rx_SetBeforeProc(service, proc)
            ((service)->beforeProc = (proc))
```

Instruct the *Rx* facility to call the procedure identified by the `proc` parameter immediately upon activation of a server thread to handle an incoming call. The specified procedure will be called with a single parameter, a pointer of type `struct rx_call`, identifying the call this thread will now be responsible for handling. The value returned by the procedure, if any, is discarded.

The `service` parameter should resolve to an object of type `struct rx_service`. The `proc` parameter should resolve to an object of type `(VOID *)()`. An instance of the *rx_SetBeforeProc()* macro itself resolves into the result of the assignment, which is an object of type `(VOID *)()`.

5.5.3.11 *rx_GetAfterProc()*

```
#define rx_GetAfterProc(service) ((service)->afterProc)
```

Return a pointer of type `(VOID *)()` to the procedure associated with the given *Rx* `service` that will be called immediately upon completion of the particular *Rx* call for which a server thread was activated. The `service` parameter should resolve to an object of type `struct rx_service`.

When an *Rx* service is first created (via a call to the *rx_NewService()* function), its `afterProc` field is set to a null pointer. See the description of the *rx_SetAfterProc()* below.

5.5.3.12 *rx_SetAfterProc()*

```
#define rx_SetAfterProc(service, proc)
            ((service)->afterProc = (proc))
```

Instruct the *Rx* facility to call the procedure identified by the `proc` parameter immediately upon completion of the particular *Rx* call for which a server thread was activated. The specified procedure will be called with a single parameter, a pointer of type `struct rx_call`, identifying the call this thread just handled. The value returned by the procedure, if any, is discarded.

The `service` parameter should resolve to an object of type `struct rx_service`. The `proc` parameter should resolve to an object of type `(VOID *)()`. An instance of the *rx_SetAfterProc()* macro itself resolves into the result of the assignment, which is an object of type `(VOID *)()`.

5.5.3.13 *rx_SetNewConnProc()*

```
#define rx_SetNewConnProc(service, proc)
            ((service)->newConnProc = (proc))
```

Instruct the *Rx* facility to call the procedure identified by the `proc` parameter as the last step in the creation of a new *Rx* server-side connection for the given `service`. The specified procedure will be called with a single parameter, a pointer of type `(struct rx_connection *)`, identifying the connection structure that was just built. The value returned by the procedure, if any, is discarded.

The `service` parameter should resolve to an object of type `struct rx_service`. The `proc` parameter should resolve to an object of type `(VOID *)()`. An instance of the *rx_SetNewConnProc()* macro itself resolves into the result of the assignment, which is an object of type `(VOID *)()`.

Note: There is no access counterpart defined for this macro, namely one that returns the current setting of a service's `newConnProc`.

5.5.3.14 *rx_SetDestroyConnProc()*

```
#define rx_SetDestroyConnProc(service, proc)
            ((service)->destroyConnProc = (proc))
```

Instruct the *Rx* facility to call the procedure identified by the `proc` parameter just before a server connection associated with the given *Rx* `service` is destroyed. The specified procedure will be called with a single parameter, a pointer of type `(struct rx_connection *)`, identifying the connection about to be destroyed. The value returned by the procedure, if any, is discarded.

The `service` parameter should resolve to an object of type `struct rx_service`. The `proc` parameter should resolve to an object of type `(VOID *)()`. An instance of the *rx_SetDestroyConnProc()* macro itself resolves into the result of the assignment, which is an object of type `(VOID *)()`.

Note: There is no access counterpart defined for this macro, namely one that returns the current setting of a service's `destroyConnProc`.

5.5.4 Security-Related Operations

The following macros are callable by *Rx* security modules, and assist in getting and setting header and trailer lengths, setting actual packet size, and finding the beginning of the security header (or data).

5.5.4.1 *rx_GetSecurityHeaderSize()*

```
#define rx_GetSecurityHeaderSize(conn) ((conn)->securityHeaderSize)
```

Generate a reference to the field in an *Rx* connection structure that records the length in bytes of the associated security module's packet header data.

The `conn` parameter should resolve to an object of type `struct rx_connection`. An instance of the *rx_GetSecurityHeaderSize()* macro itself resolves into an object of type `u_short`.

5.5.4.2 *rx_SetSecurityHeaderSize()*

```
#define rx_SetSecurityHeaderSize(conn, length)
            ((conn)->securityHeaderSize = (length))
```

Set the field in a connection structure that records the length in bytes of the associated security module's packet header data.

The `conn` parameter should resolve to an object of type `struct rx_connection`. The `length` parameter should resolve to an object of type `u_short`. An instance of the *rx_SetSecurityHeaderSize()* macro itself resolves into the result of the assignment, which is an object of type `u_short`.

5.5.4.3 *rx_GetSecurityMaxTrailerSize()*

```
#define rx_GetSecurityMaxTrailerSize(conn)
            ((conn)->securityMaxTrailerSize)
```

Generate a reference to the field in an *Rx* connection structure that records the maximum length in bytes of the associated security module's packet trailer data.

The `conn` parameter should resolve to an object of type `struct rx_connection`. An instance of the *rx_GetSecurityMaxTrailerSize()* macro itself resolves into an object of type `u_short`.

5.5.4.4 *rx_SetSecurityMaxTrailerSize()*

```
#define rx_SetSecurityMaxTrailerSize(conn, length)
            ((conn)->securityMaxTrailerSize = (length))
```

Set the field in a connection structure that records the maximum length in bytes of the associated security module's packet trailer data.

The `conn` parameter should resolve to an object of type `struct rx_connection`. The `length` parameter should resolve to an object of type `u_short`. An instance of the *rx_SetSecurityHeaderSize()* macro itself resolves into the result of the assignment, which is an object of type `u_short`.

5.5.5 Sizing Operations

The macros described in this section assist the application programmer in determining the sizes of the various *Rx* packet regions, as well as their placement within a packet buffer.

Programming Interface

5.5.5.1 *rx_UserDataOf()*

```
#define rx_UserDataOf(conn, packet)
            (((char *) (packet)->wire.data) +
                       (conn)->securityHeaderSize)
```

Generate a pointer to the beginning of the actual user data in the given *Rx* `packet`, that is associated with the connection described by the `conn` pointer. User data appears immediately after the packet's security header region, whose length is determined by the security module used by the connection. The `conn` parameter should resolve to an object of type `struct rx_connection`. The `packet` parameter should resolve to an object of type `struct rx_packet`. An instance of the *rx_UserDataOf()* macro itself resolves into an object of type `(char *)`.

5.5.5.2 *rx_MaxUserDataSize()*

```
#define rx_MaxUserDataSize(conn)
            ((conn)->peer->packetSize
             - RX_HEADER_SIZE
             - (conn)->securityHeaderSize
             - (conn)->securityMaxTrailerSize)
```

Return the maximum number of user data bytes that may be carried by a packet on the *Rx* connection described by the `conn` pointer. The overall packet size is reduced by the IP, UDP, and *Rx* headers, as well as the header and trailer areas required by the connection's security module.

The `conn` parameter should resolve to an object of type `struct rx_connection`. An instance of the *rx_MaxUserDataSize()* macro itself resolves into the an object of type `(u_short)`.

5.5.6 Complex Operations

Two *Rx* macros are designed to handle potentially complex operations, namely reading data from an active incoming call and writing data to an active outgoing call. Each call structure has an internal buffer that is used to collect and cache data traveling through the call. This buffer is used in conjunction with reading or writing to the actual *Rx* packets traveling on the wire in support of the call. The *rx_Read()* and *rx_Write()* macros allow their caller to simply manipulate the internal data buffer associated with the *Rx* call structures whenever possible, thus avoiding the overhead associated with a function call. When buffers are either filled or drained (depending on the direction of the data flow), these macros will then call functions to handle the more complex cases of generating or receiving packets in support of the operation.

5.5.6.1 *rx_Read()*

```
#define rx_Read(call, buf, nbytes)
    ((call)->nLeft > (nbytes) ?
        bcopy((call)->bufPtr, (buf), (nbytes)),
        (call)->nLeft -= (nbytes), (call)->bufPtr += (nbytes), (nbytes)
        : rx_ReadProc((call), (buf), (nbytes)))
```

Read `nbytes` of data from the given *Rx* `call` into the buffer to which `buf` points. If the call's internal buffer has at least `nbytes` bytes already filled, then this is done in-line with a copy and some pointer and counter updates within the call structure. If the `call`'s internal buffer doesn't have enough data to satisfy the request, then the *rx_ReadProc()* function will handle this more complex situation.

In either case, the *rx_Read()* macro returns the number of bytes actually read from the call, resolving to an object of type `int`. If *rx_Read()* returns fewer than `nbytes` bytes, the call status should be checked via the *rx_Error()* macro.

5.5.6.2 *rx_Write()*

```
#define rx_Write(call, buf, nbytes)
    ((call)->nFree > (nbytes) ?
        bcopy((buf), (call)->bufPtr, (nbytes)),
        (call)->nFree -= (nbytes),
        (call)->bufPtr += (nbytes), (nbytes)
        : rx_WriteProc((call), (buf), (nbytes)))
```

Write `nbytes` of data from the buffer pointed to by `buf` into the given *Rx* `call`. If the call's internal buffer has at least `nbytes` bytes free, then this is done in-line with a copy and some pointer and counter updates within the call structure. If the `call`'s internal buffer doesn't have room, then the *rx_WriteProc()* function will handle this more complex situation.

In either case, the *rx_Write()* macro returns the number of bytes actually written to the call, resolving to an object of type `int`. If zero is returned, the call status should be checked via the *rx_Error()* macro.

5.5.7 Security Operation Invocations

Every *Rx* security module is required to implement an identically-named set of operations, through which the security mechanism it defines is invoked. This characteristic interface is reminiscent of the vnode interface defined and popularized for file systems by Sun Microsystems [8]. The structure defining this function array is described in Section 5.3.1.1.

These security operations are part of the `struct rx_securityClass`, which keeps not only the ops array itself but also any private data they require and a reference count. Every *Rx* service contains an array of these security class objects, specifying the range of security mechanisms it is capable of enforcing. Every *Rx* connection within a service is associated with exactly one of that service's security objects, and every call issued on the connection will execute the given security protocol.

The macros described below facilitate the execution of the security module interface functions. They are covered in the same order they appear in the `struct rx_securityOps` declaration.

5.5.7.1 *RXS_OP()*

```
#if defined(__STDC__) && !defined(__HIGHC__)
    #define RXS_OP(obj, op, args)
        ((obj->ops->op_ ## op) ? (*(obj)->ops->op_ ## op)args : 0)
#else
    #define RXS_OP(obj, op, args)
        ((obj->ops->op_/**/op) ? (*(obj)->ops->op_/**/op)args : 0)
#endif
```

The *RXS_OP* macro represents the workhorse macro in this group, used by all the others. It takes three arguments, the first of which is a pointer to the security object to be referenced. This `obj` parameter must resolve to an object of type `(struct rx_securityOps *)`. The second parameter identifies the specific `op` to be performed on this security object. The actual text of this `op` argument is used to name the desired opcode function. The third and final argument, `args`, specifies the text of the argument list to be fed to the chosen security function. Note that this argument *must* contain the bracketing parentheses for the function call's arguments. In fact, note that each of the security function access macros defined below provides the enclosing parentheses to this third *RXS_OP()* macro.

5.5.7.2 *RXS_Close()*

```
#define RXS_Close(obj) RXS_OP(obj, Close, (obj))
```

This macro causes the execution of the interface routine occupying the *op_Close()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* immediately before a security object is discarded. Among the responsibilities of such a function might be decrementing the object's `refCount` field, and thus perhaps freeing up any space contained within the security object's private storage region, referenced by the object's `privateData` field.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. In generating a call to the security object's *op_Close()* routine, the `obj` pointer is used as its single parameter. An invocation of the *RXS_Close()* macro results in a return value identical to that of the *op_Close()* routine, namely a value of type `int`.

5.5.7.3 *RXS_NewConnection()*

```
#define RXS_NewConnection(obj, conn)
          RXS_OP(obj, NewConnection, (obj, conn))
```

This macro causes the execution of the interface routine in the *op_NewConnection()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* immediately after a connection using the given security object is created. Among the responsibilities of such a function might be incrementing the object's `refCount` field, and setting any per-connection information based on the associated security object's private storage region, as referenced by the object's `privateData` field.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the newly-created connection structure, and must resolve into an object of type `(struct rx_connection *)`.

In generating a call to the routine located at the security object's *op_NewConnection()* slot, the `obj` and `conn` pointers are used as its two parameters. An invocation of the *RXS_NewConnection()* macro results in a return value identical to that of the *op_NewConnection()* routine, namely a value of type `int`.

5.5.7.4 *RXS_PreparePacket()*

```
#define RXS_PreparePacket(obj, call, packet)
```

```
            RXS_OP(obj, PreparePacket, (obj, call, packet))
```

This macro causes the execution of the interface routine in the *op_PreparePacket()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time it prepares an outward-bound packet. Among the responsibilities of such a function might be computing information to put into the packet's security header and/or trailer.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `call` argument contains a pointer to the *Rx* call to which the given packet belongs, and must resolve to an object of type `(struct rx_call *)`. The final argument, `packet`, contains a pointer to the packet itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_PreparePacket()* slot, the `obj`, `call`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_PreparePacket()* macro results in a return value identical to that of the *op_PreparePacket()* routine, namely a value of type `int`.

5.5.7.5 *RXS_SendPacket()*

```
#define RXS_SendPacket(obj, call, packet)
            RXS_OP(obj, SendPacket, (obj, call, packet))
```

This macro causes the execution of the interface routine occupying the *op_SendPacket()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time it physically transmits an outward-bound packet. Among the responsibilities of such a function might be recomputing information in the packet's security header and/or trailer.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `call` argument contains a pointer to the *Rx* call to which the given packet belongs, and must resolve to an object of type `(struct rx_call *)`. The final argument, `packet`, contains a pointer to the packet itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_SendPacket()* slot, the `obj`, `call`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_SendPacket()* macro results in a return value identical to that of the *op_SendPacket()* routine, namely a value of type `int`.

5.5.7.6 *RXS_CheckAuthentication()*

```
#define RXS_CheckAuthentication(obj, conn)
            RXS_OP(obj, CheckAuthentication, (obj, conn))
```

This macro causes the execution of the interface routine in the *op_CheckAuthentication()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time it needs to check whether the given connection is one on which authenticated calls are being performed. Specifically, a value of 0 is returned if authenticated calls are *not* being executed on this connection, and a value of 1 is returned if they are.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection checked as to whether authentication is being performed, and must resolve to an object of type `(struct rx_connection *)`.

In generating a call to the routine in the security object's *op_CheckAuthentication()* slot, the `obj` and `conn` pointers are used as its two parameters. An invocation of the *RXS_CheckAuthentication()* macro results in a return value identical to that of the *op_CheckAuthentication()* routine, namely a value of type `int`.

5.5.7.7 *RXS_CreateChallenge()*

```
#define RXS_CreateChallenge(obj, conn)
            RXS_OP(obj, CreateChallenge, (obj, conn))
```

This macro causes the execution of the interface routine in the *op_CreateChallenge()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time a challenge event is constructed for a given connection. Among the responsibilities of such a function might be marking the connection as temporarily unauthenticated until the given challenge is successfully met.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection for which the authentication challenge is being constructed, and must resolve to an object of type `(struct rx_connection *)`.

In generating a call to the routine located at the security object's *op_CreateChallenge()* slot, the `obj` and `conn` pointers are used as its two parameters. An invocation of the *RXS_CreateChallenge()* macro results in a return value identical to that of the *op_CreateChallenge()* routine, namely a value of type `int`.

5.5.7.8 *RXS_GetChallenge()*

```
#define RXS_GetChallenge(obj, conn, packet)
          RXS_OP(obj, GetChallenge, (obj, conn, packet))
```

This macro causes the execution of the interface routine occupying the *op_GetChallenge()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time a challenge packet is constructed for a given connection. Among the responsibilities of such a function might be constructing the appropriate challenge structures in the area of packet dedicated to security matters.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection to which the given challenge packet belongs, and must resolve to an object of type `(struct rx_connection *)`. The final argument, `packet`, contains a pointer to the challenge packet itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_GetChallenge()* slot, the `obj`, `conn`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_GetChallenge()* macro results in a return value identical to that of the *op_GetChallenge()* routine, namely a value of type `int`.

5.5.7.9 *RXS_GetResponse()*

```
#define RXS_GetResponse(obj, conn, packet)
          RXS_OP(obj, GetResponse, (obj, conn, packet))
```

This macro causes the execution of the interface routine occupying the *op_GetResponse()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* on the server side each time a response to a challenge packet must be received.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* client connection that must respond to the authentication challenge, and must resolve to a `(struct rx_connection *)` object. The final argument, `packet`, contains a pointer to the packet to be built in response to the challenge. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_GetResponse()* slot, the `obj`, `conn`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_GetResponse()* macro results in a return value identical to that of the *op_GetResponse()* routine, namely a value of type `int`.

5.5.7.10 *RXS_CheckResponse()*

```
#define RXS_CheckResponse(obj, conn, packet)
            RXS_OP(obj, CheckResponse, (obj, conn, packet))
```

This macro causes the execution of the interface routine in the *op_CheckResponse()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* on the server side each time a response to a challenge packet is received for a given connection. The responsibilities of such a function might include verifying the integrity of the response, pulling out the necessary security information and storing that information within the affected connection, and otherwise updating the state of the connection.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* server connection to which the given challenge response is directed. This argument must resolve to an object of type `(struct rx_connection *)`. The final argument, `packet`, contains a pointer to the packet received in response to the challenge itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_CheckResponse()* slot, the `obj`, `conn`, and `packet` pointers are ued as its three parameters. An invocation of the *RXS_CheckResponse()* macro results in a return value identical to that of the *op_CheckResponse()* routine, namely a value of type `int`.

5.5.7.11 *RXS_CheckPacket()*

```
#define RXS_CheckPacket(obj, call, packet)
            RXS_OP(obj, CheckPacket, (obj, call, packet))
```

This macro causes the execution of the interface routine occupying the *op_CheckPacket()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time a packet is received. The responsibilities of such a function might include verifying the integrity of given packet, detecting any unauthorized modifications or tampering.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection to which the given challenge response is directed, and must resolve to an object of type `(struct rx_connection *)`. The final argument, `packet`, contains a pointer to the packet received in response to the challenge itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_CheckPacket()* slot, the `obj`, `conn`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_CheckPacket()* macro results in a return value identical to that of the *op_CheckPacket()* routine, namely a value of type `int`.

Please note that any non-zero return will cause *Rx* to abort all calls on the connection. Furthermore, the connection itself will be marked as being in error in such a case, causing it to reject any further incoming packets.

5.5.7.12 *RXS_DestroyConnection()*

```
#define RXS_DestroyConnection(obj, conn)
            RXS_OP(obj, DestroyConnection, (obj, conn))
```

This macro causes the execution of the interface routine in the *op_DestroyConnection()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time a connection employing the given security object is being destroyed. The responsibilities of such a function might include deleting any private data maintained by the security module for this connection.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection being reaped, and must resolve to a `(struct rx_connection *)` object.

In generating a call to the routine located at the security object's *op_DestroyConnection()* slot, the `obj` and `conn` pointers are used as its two parameters. An invocation of the *RXS_DestroyConnection()* macro results in a return value identical to that of the *op_DestroyConnection()* routine, namely a value of type `int`.

5.5.7.13 *RXS_GetStats()*

```
#define RXS_GetStats(obj, conn, stats)
            RXS_OP(obj, GetStats, (obj, conn, stats))
```

This macro causes the execution of the interface routine in the *op_GetStats()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time current statistics concerning the given security object are desired.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection using the security object to be examined, and must resolve to an object of type `(struct rx_connection *)`. The final argument, `stats`, contains a pointer to a region to be filled with the desired statistics. It should resolve to an object of type `(struct rx_securityObjectStats *)`.

In generating a call to the routine located at the security object's *op_GetStats()* slot, the `obj`, `conn`, and `stats` pointers are used as its three parameters. An invocation of the *RXS_GetStats()* macro results in a return value identical to that of the *op_GetStats()* routine, namely a value of type `int`.

5.6 Functions

Rx exports a collection of functions that, in conjuction with the macros explored in Section 5.5, allows its clients to set up and export services, create and tear down connections to these services, and execute remote procedure calls along these connections.

This paper employs two basic categorizations of these *Rx* routines. One set of functions is meant to be called directly by clients of the facility, and are referred to as the *exported operations*. The individual members of the second set of functions are *not* meant to be called directly by *Rx* clients, but rather are called by the collection of defined macros, so they must still be lexically visible. These indirectly-executed routines are referred to here as the *semi-exported operations*.

All *Rx* routines return zero upon success. The range of error codes employed by *Rx* is defined in Section 5.2.15.

5.6.1 Exported Operations

5.6.2 rx_Init — Initialize *Rx*

`int` **rx_Init**`(IN int port)`

Description

Initialize the *Rx* facility. If a non-zero `port` number is provided, it becomes the default port number for any service installed later. If 0 is provided for the `port`, a random port will be chosen by the system. The *rx_Init()* function sets up internal tables and timers, along with starting up the listener thread.

Error Codes

`RX_ADDRINUSE` The `port` provided has already been taken.

5.6.3 rx_NewService — Create and install a new service

```
struct rx_service *rx_NewService(IN u_short port;
                                 IN u_short serviceId;
                                 IN char *serviceName;
                                 IN struct rx_securityClass **securityObjects;
                                 IN int nSecurityObjects;
                                 IN long (*serviceProc)())
```

Description

Create and advertise a new *Rx* service. A service is uniquely named by a UDP `port` number plus a non-zero 16-bit `serviceId` on the given host. The `port` argument may be set to zero if *rx_Init()* was called with a non-zero port number, in which case that original port will be used. A `serviceName` must also be provided, to be used for identification purposes (e.g., the service name might be used for probing for statistics). A pointer to an array of `nSecurityObjects` security objects to be associated with the new service is given in . `securityObjects`. The service's *executeRequestProc()* pointer is set to `serviceProc`.

The function returns a pointer to a descriptor for the requested *Rx* service. A null return value indicates that the new service could not be created. Possible reasons include:

- The `serviceId` parameter was found to be zero.

- A port value of zero was specified at *Rx* initialization time (i.e., when *rx_init()* was called), requiring a non-zero value for the `port` parameter here.
- Another *Rx* service is already using `serviceId`.
- *Rx* has already created the maximum `RX_MAX_SERVICES` *Rx* services (see Section 5.2.1).

Error Codes

`(struct rx_service *) NULL` The new *Rx* service could not be created, due to one of the errors listed above.

5.6.4 rx_NewConnection — Create a new connection to a given service

```
struct rx_connection *rx_NewConnection(IN u_long shost,
                                       IN u_short sport,
                                       IN u_short sservice,
                                       IN struct rx_securityClass *securityObject,
                                       IN int service SecurityIndex)
```

Description

Create a new *Rx* client connection to service `sservice` on the host whose IP address is contained in `shost` and to that host's `sport` UDP port. The corresponding *Rx* service identifier is expected in `sservice`. The caller also provides a pointer to the security object to use for the connection in `securityObject`, along with that object's `serviceSecurityIndex` among the security objects associated with service `sservice` via a previous *rx_NewService()* call (see Section 5.6.3).

Note: It is permissible to provide a null value for the `securityObject` parameter if the chosen `serviceSecurityIndex` is zero. This corresponds to the pre-defined null security object, which does not engage in authorization checking of any kind.

Error Codes

--- A pointer to an initialized *Rx* connection is always returned, unless *osi_Panic()* is called due to memory allocation failure.

5.6.5 rx_NewCall — Start a new call on the given connection

```
struct rx_call *rx_NewCall(IN struct rx_connection *conn)
```

Description

Start a new *Rx* remote procedure call on the connection specified by the `conn` parameter. The existing call structures (up to `RX_MAXCALLS` of them) are examined in order. The first non-active call encountered (i.e., either unused or whose `call->state` is `RX_STATE_DALLY`) will be appropriated and reset if necessary. If all call structures are in active use, the `RX_CONN_MAKECALL_WAITING` flag is set in the `conn->flags` field, and the thread handling this request will sleep until a call structure comes free. Once a call structure has been reserved, the keep-alive protocol is enabled for it.

The state of the given connection determines the detailed behavior of the function. The `conn->timeout` field specifies the absolute upper limit of the number of seconds this particular call may be in operation. After this time interval, calls to such routines as *rx_SendData()* or *rx_ReadData()* will fail with an `RX_CALL_TIMEOUT` indication.

Error Codes

--- A pointer to an initialized *Rx* call is always returned, unless *osi_Panic()* is called due to memory allocation failure.

5.6.6 rx_EndCall — Terminate the given call

```
int rx_EndCall(IN struct rx_call *call,
               IN long rc)
```

Description

Indicate that the *Rx* call described by the structure located at `call` is finished, possibly prematurely. The value passed in the `rc` parameter is returned to the peer, if appropriate. The final error code from processing the call will be returned as *rx_EndCall()*'s value. The given call's state will be set to `RX_STATE_DALLY`, and threads waiting to establish a new call on this connection are signalled (see the description of the *rx_NewCall()* in Section 5.6.5).

Error Codes

`-1` Unspecified error has occurred.

5.6.7 rx_StartServer — Activate installed *rx* service(s)

```
void rx_StartServer(IN int donateMe)
```

Description

This function starts server threads in support of the *Rx* services installed via calls to *rx_NewService()* (see Section 5.6.3). This routine first computes the number of server threads it must create, governed by the `minProcs` and `maxProcs` fields in the installed service descriptors. The `minProcs` field specifies the minimum number of threads that are guaranteed to be concurrently available to the given service. The `maxProcs` field specifies the maximum number of threads that may ever be concurrently assigned to the particular service, if idle threads are available. Using this information, *rx_StartServer()* computes the correct overall number of threads as follows: For each installed service, `minProcs` threads will be created, enforcing the minimality guarantee. Calculate the maximum difference between the `maxProcs` and `minProcs` fields for each service, and create this many additional server threads, enforcing the maximality guarantee.

If the value placed in the `donateMe` argument is zero, then *rx_StartServer()* will simply return after performing as described above. Otherwise, the thread making the *rx_StartServer()* call will itself begin executing the server thread loop. In this case, the `rx_StartServer()` call will never return.

Error Codes

`---` None.

5.6.8 rx_PrintStats — Print basic statistics to a file

`void` **rx_PrintStats**`(IN FILE *file)`

Description

Prints *Rx* statistics (basically the contents of the `struct rx_stats` holding the statistics for the *Rx* facility) to the open file descriptor identified by `file`. The output is ASCII text, and is intended for human consumption.

Note: This function is available only if the *Rx* package has been compiled with the `RXDEBUG` flag.

Error Codes

`---` None.

5.6.9 rx_PrintPeerStats — Print peer statistics to a file

```
void rx_PrintPeerStats(IN FILE *file,
                       IN struct rx_peer *peer)
```

Description

Prints the *Rx* peer statistics found in `peer` to the open file descriptor identified by `file`. The output is in normal ASCII text, and is intended for human consumption.

Note: This function is available only if the *Rx* package has been compiled with the `RXDEBUG` flag.

Error Codes

--- None.

5.6.10 rx_Finalize — Shut down *Rx* gracefully

```
void rx_Finalize()
```

Description

This routine may be used to shut down the *Rx* facility for either server or client applications. All of the client connections will be gracefully garbage-collected after their active calls are cleaned up. The result of calling *rx_Finalize()* from a client program is that the server-side entity will be explicitly advised that the client has terminated. This notification frees the server-side application from having to probe the client until its records eventually time out, and also allows it to free resources currently assigned to that client's support.

Error Codes

`---` None.

5.6.11 Semi-Exported Operations

As described in the introductory text in Section 5.6, entries in this lexically-visible set of *Rx* functions are *not* meant to be called directly by client applications, but rather are invoked by *Rx* macros called by users.

5.6.12 rx_WriteProc — Write data to an outgoing call

```
int rx_WriteProc(IN struct rx_call *call,
                 IN char *buf,
                 IN int nbytes)
```

Description

Write `nbytes` of data from buffer `buf` into the *Rx* call identified by the `call` parameter. The value returned by *rx_WriteProc()* reports the number of bytes actually written into the call. If zero is returned, then the *rx_Error()* macro may be used to obtain the call status.

This routine is called by the *rx_Write()* macro, which is why it must be exported by the *Rx* facility.

Error Codes

`0` Indicates error in the given *Rx* call; use the *rx_Error()* macro to determine the call status.

5.6.13 rx_ReadProc — Read data from an incoming call

```
int rx_ReadProc(IN struct rx_call *call,
                IN char *buf,
                IN int nbytes)
```

Description

Read up to `nbytes` of data from the *Rx* call identified by the `call` parameter into the `buf` buffer. The value returned by *rx_ReadProc()* reports the number of bytes actually read from the call. If zero is returned, then the *rx_Error()* macro may be used to obtain the call status.

This routine is called by the *rx_Read()* macro, which is why it must be exported by the *Rx* facility.

Error Codes

`0` Indicates error in the given *Rx* call; use the *rx_Error()* macro to determine the call status.

5.6.14 rx_FlushWrite — Flush buffered data on outgoing call

```
void rx_FlushWrite(IN struct rx_call *call)
```

Description

Flush any buffered data on the given *Rx* call to the stream. If the call is taking place on a server connection, the `call->mode` is set to `RX_MODE_EOF`. If the call is taking place on a client connection, the `call->mode` is set to `RX_MODE_RECEIVING`.

Error Codes

--- None.

5.6.15 rx_SetArrivalProc — Set function to invoke upon call packet arrival

```
void rx_SetArrivalProc(IN struct rx_call *call,
                       IN VOID (*proc)(),
                       IN VOID *handle,
                       IN VOID *arg)
```

Description

Establish a procedure to be called when a packet arrives for a call. This routine will be called at most once after each call, and will also be called if there is an error condition on the call or the call is complete. The *rx_SetArrivalProc()* function is used by multicast *Rx* routines to build a selection function that determines which of several calls is likely to be a good one to read from. The implementor's comments in the *Rx* code state that, due to the current implementation, it is probably only reasonable to use *rx_SetArrivalProc()* immediately after an *rx_NewCall()*, and to only use it once.

Error Codes

--- None.

Chapter 6

Example Server and Client

6.1 Introduction

This chapter provides a sample program showing the use of *Rx*. Specifically, the *rxdemo* application, with all its support files, is documented and examined. The goal is to provide the reader with a fully-developed and operational program illustrating the use of both regular *Rx* remote procedure calls and streamed RPCs. The full text of the *rxdemo* application is reproduced in the sections below, along with additional commentary.

Readers wishing to directly experiment with this example *Rx* application are encouraged to examine the on-line version of *rxdemo*. Since it is a program of general interest, it has been installed in the **usr/contrib** tree in the `grand.central.org` cell. This area contains user-contributed software for the entire AFS community. At the top of this tree is the */afs/grand.central.org/darpa/usr/contrib* directory. Both the server-side and client-side *rxdemo* binaries (*rxdemo_server* and *rxdemo_client*, respectively) may be found in the *bin* subdirectory. The actual sources reside in the *.site/grand.central.org/rxdemo/src* subdirectory.

The *rxdemo* code is composed of two classes of files, namely those written by a human programmer and those generated from the human-written code by the *Rxgen* tool. Included in the first group of files are:

- *rxdemo.xg* This is the RPC interface definition file, providing high-level definitions of the supported calls.
- *rxdemo_client.c*: This is the *rxdemo* client program, calling upon the associated server to perform operations defined by *rxdemo.xg*.

- *rxdemo_server.c*: This is the *rxdemo* server program, implementing the operations promised in *rxdemo.xg*.
- Makefile: This is the file that directs the compilation and installation of the *rxdemo* code.

The class of automatically-generated files includes the following items:

- *rxdemo.h*: This header file contains the set of constant definitions present in *rxdemo.xg*, along with information on the RPC opcodes defined for this *Rx* service.
- *rxdemo.cs.c*: This client-side stub file performs all the marshalling and unmarshalling of the arguments for the RPC routines defined in *rxdemo.xg*.
- *rxdemo.ss.c*: This stub file similarly defines all the marshalling and unmarshalling of arguments for the server side of the RPCs, invokes the routines defined within *rxdemo_server.c* to implement the calls, and also provides the dispatcher function.
- *rxdemo.xdr.c*: This module defines the routines required to convert complex user-defined data structures appearing as arguments to the *Rx* RPC calls exported by *rxdemo.xg* into network byte order, so that correct communication is guaranteed between clients and server with different memory organizations.

The chapter concludes with a section containing sample output from running the *rxdemo* server and client programs.

6.2 Human-Generated Files

The *rxdemo* application is based on the four human-authored files described in this section. They provide the basis for the construction of the full set of modules needed to implement the specified *Rx* service.

6.2.1 Interface File: *rxdemo.xg*

This file serves as the RPC interface definition file for this application. It defines various constants, including the *Rx* service port to use and the index of the null security object (no encryption is used by *rxdemo*). It defines the `RXDEMO_MAX` and `RXDEMO_MIN` constants, which will be used by the server as the upper and lower bounds on the number of *Rx* listener threads to run. It also defines the set of error codes exported by this facility. Finally, it provides the RPC function declarations, namely *Add()* and *GetFile()*. Note that when building the actual function definitions, *Rxgen* will prepend the value of the **package** line in this file, namely "RXDEMO_", to the function declarations. Thus, the generated functions become *RXDEMO_Add()* and *RXDEMO_GetFile()*, respectively. Note the use of the `split` keyword in the *RXDEMO_GetFile()* declaration, which specifies that this is a streamed call, and actually generates *two* client-side stub routines (see Section 6.3.1).

```
/*=======================================================================
* Interface for an example Rx server/client application, using both     *
* standard and streamed calls.                                          *
*                                                                       *
* Edward R. Zayas                                                       *
* Transarc Corporation                                                  *
*                                                                       *
*                                                                       *
* The United States Government has rights in this work pursuant         *
* to contract no. MDA972-90-C-0036 between the United States Defense    *
* Advanced Research Projects Agency and Transarc Corporation.           *
*                                                                       *
* (C) Copyright 1991 Transarc Corporation                               *
*                                                                       *
* Redistribution and use in source and binary forms are permitted       *
* provided that: (1) source distributions retain this entire copy-      *
* right notice and comment, and (2) distributions including binaries   *
* display the following acknowledgement:                                *
*                                                                       *
*       ''This product includes software developed by Transarc         *
*       Corporation and its contributors''                              *
*                                                                       *
* in the documentation or other materials mentioning features or        *
* use of this software.  Neither the name of Transarc nor the names     *
* of its contributors may be used to endorse or promote products        *
* derived from this software without specific prior written             *
* permission.                                                           *
*                                                                       *
* THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  *
* WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  *
* MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                 *
*=======================================================================*/

package RXDEMO_

%#include <rx/rx.h>
%#include <rx/rx_null.h>

%#define RXDEMO_SERVER_PORT     8000    /*Service port to advertise*/
%#define RXDEMO_SERVICE_PORT       0    /*User server's port*/
%#define RXDEMO_SERVICE_ID         4    /*Service ID*/
%#define RXDEMO_NULL_SECOBJ_IDX    0    /*Index of null security object*/
```

```
/*
 * Maximum number of requests that will be handled by this service
 * simultaneously.  This number will be guaranteed to execute in
 * parallel if other service's results are being processed.
 */
%#define RXDEMO_MAX     3

/*
 * Minimum number of requests that are guaranteed to be handled
 * simultaneously.
 */
%#define RXDEMO_MIN     2

/*
 * Index of the "null" security class in the sample service.
 */
%#define RXDEMO_NULL    0

/*
 * Maximum number of characters in a file name (for demo purposes).
 */
%#define RXDEMO_NAME_MAX_CHARS  64

/*
 * Define the max number of bytes to transfer at one shot.
 */
%#define RXDEMO_BUFF_BYTES 512


/*
 * Values returned by the RXDEMO_GetFile() call.
 *      RXDEMO_CODE_SUCCESS     : Everything went fine.
 *      RXDEMO_CODE_CANT_OPEN   : Can't open named file.
 *      RXDEMO_CODE_CANT_STAT   : Can't stat open file.
 *      RXDEMO_CODE_CANT_READ   : Error reading the open file.
 *      RXDEMO_CODE_WRITE_ERROR : Error writing the open file.
 */
%#define RXDEMO_CODE_SUCCESS     0
%#define RXDEMO_CODE_CANT_OPEN   1
%#define RXDEMO_CODE_CANT_STAT   2
%#define RXDEMO_CODE_CANT_READ   3
%#define RXDEMO_CODE_WRITE_ERROR 4

/*
 * ------------ Interface calls defined for this service ------------
 */


/*---------------------------------------------------------------------
 * RXDEMO_Add
 *
 * Summary:
 *      Add the two numbers provided and return the result.
```

```
 *
 * Parameters:
 *      int a_first   : First operand.
 *      int a_second  : Second operand.
 *      int *a_result : Sum of the above.
 *
 * Side effects:
 *      None.
 *------------------------------------------------------------------------*/

Add(IN  int a,
        int b,
    OUT int *result) = 1;

/*------------------------------------------------------------------------
 * RXDEMO_GetFile
 *
 * Summary:
 *      Return the contents of the named file in the server's
 *      environment.
 *
 * Parameters:
 *      STRING a_nameToRead : Name of the file whose contents are to be
 *                            fetched.
 *      int *a_result       : Set to the result of opening and reading the
 *                            file on the server side.
 *
 * Side effects:
 *      None.
 *------------------------------------------------------------------------*/

GetFile(IN string a_nameToRead<RXDEMO_NAME_MAX_CHARS>,
        OUT int *a_result) split = 2;
```

6.2.2 Client Program: *rxdemo_client.c*

The *rxdemo* client program, *rxdemo_client*, calls upon the associated server to perform operations defined by *rxdemo.xg*. After its header, it defines a private *GetIPAddress()* utility routine, which given a character string host name will return its IP address.

```
/*=========================================================================
% Client side of an example Rx application, using both standard and     %
% streamed calls.                                                         %
%                                                                         %
% Edward R. Zayas                                                         %
% Transarc Corporation                                                    %
%                                                                         %
%                                                                         %
```

```
% The United States Government has rights in this work pursuant        %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                      %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                      %
% Redistribution and use in source and binary forms are permitted      %
% provided that: (1) source distributions retain this entire copy-     %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                      %
%       ``This product includes software developed by Transarc         %
%       Corporation and its contributors''                             %
%                                                                      %
% in the documentation or other materials mentioning features or       %
% use of this software.  Neither the name of Transarc nor the names    %
% of its contributors may be used to endorse or promote products       %
% derived from this software without specific prior written            %
% permission.                                                          %
%                                                                      %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                %
%=====================================================================*/

#include <sys/types.h>
#include <netdb.h>
#include <stdio.h>
#include "rxdemo.h"

static char pn[] = "rxdemo";    /*Program name*/

static u_long GetIpAddress(a_hostName)
    char *a_hostName;

{ /*GetIPAddress*/

    static char rn[] = "GetIPAddress";  /*Routine name*/
    struct hostent *hostEntP;           /*Ptr to host descriptor*/
    u_long hostIPAddr;                  /*Host IP address*/

    hostEntP = gethostbyname(a_hostName);
    if (hostEntP == (struct hostent *)0) {
        printf("[%s:%s] Host '%s' not found\n",
               pn, rn, a_hostName);
        exit(1);
    }
    if (hostEntP->h_length != sizeof(u_long)) {
        printf("[%s:%s] Wrong host address length (%d bytes instead of %d)",
               pn, rn, hostEntP->h_length, sizeof(u_long));
        exit(1);
    }
    bcopy(hostEntP->h_addr, (char *)&hostIPAddr, sizeof(hostIPAddr));
    return(hostIPAddr);
```

```
} /*GetIpAddress*/
```

The `main` program section of the client code, after handling its command line arguments, starts off by initializing the *Rx* facility.

```
main(argc, argv)
    int argc;
    char **argv;

{ /*Main*/

    struct rx_connection *rxConnP;        /*Ptr to server connection*/
    struct rx_call *rxCallP;              /*Ptr to Rx call descriptor*/
    u_long hostIPAddr;                    /*IP address of chosen host*/
    int demoUDPPort;                      /*UDP port of Rx service*/
    struct rx_securityClass *nullSecObjP; /*Ptr to null security object*/
    int operand1, operand2;               /*Numbers to add*/
    int sum;                              /*Their sum*/
    int code;                             /*Return code*/
    char fileName[64];                    /*Buffer for desired file's name*/
    long fileDataBytes;                   /*Num bytes in file to get*/
    char buff[RXDEMO_BUFF_BYTES+1];       /*Read buffer*/
    int currBytesToRead;                  /*Num bytes to read in one iteration*/
    int maxBytesToRead;                   /*Max bytes to read in one iteration*/
    int bytesReallyRead;                  /*Num bytes read off Rx stream*/
    int getResults;                       /*Results of the file fetch*/

    printf("\n%s: Example Rx client process\n\n", pn);
    if ((argc < 2) || (argc > 3)) {
        printf("Usage: rxdemo <HostName> [PortToUse]");
        exit(1);
    }

    hostIPAddr = GetIpAddress(argv[1]);
    if (argc > 2)
        demoUDPPort = atoi(argv[2]);
    else
        demoUDPPort = RXDEMO_SERVER_PORT;

    /*
     * Initialize the Rx facility.
     */
    code = rx_Init(htons(demoUDPPort));
    if (code) {
        printf("** Error calling rx_Init(); code is %d\n",
               code);
        exit(1);
    }

    /*
     * Create a client-side null security object.
```

```
     */
    nullSecObjP = rxnull_NewClientSecurityObject();
    if (nullSecObjP == (struct rx_securityClass *)0) {
        printf("%s: Can't create a null client-side security object!\n",
               pn);
        exit(1);
    }

    /*
     * Set up a connection to the desired Rx service, telling it to use
     * the null security object we just created.
     */
    printf("Connecting to Rx server on `%s', IP address 0x%x, UDP port %d\n",
           argv[1], hostIPAddr, demoUDPPort);
    rxConnP = rx_NewConnection(hostIPAddr,
                               RXDEMO_SERVER_PORT,
                               RXDEMO_SERVICE_ID,
                               nullSecObjP,
                               RXDEMO_NULL_SECOBJ_IDX);
    if (rxConnP == (struct rx_connection *)0) {
        printf("rxdemo: Can't create connection to server!\n");
        exit(1);
    }
    else
        printf("  ---> Connected.\n");
```

The *rx_Init()* invocation initializes the *Rx* library and defines the desired service UDP port (in network byte order). The *rxnull_NewClientSecurityObject()* call creates a client-side *Rx* security object that does not perform any authentication on *Rx* calls. Once a client authentication object is in hand, the program calls *rx_NewConnection()*, specifying the host, UDP port, *Rx* service ID, and security information needed to establish contact with the *rxdemo_server* entity that will be providing the service.

With the *Rx* connection in place, the program may perform RPCs. The first one to be invoked is *RXDEMO_Add()*:

```
    /*
     * Perform our first, simple remote procedure call.
     */
    operand1 = 1; operand2 = 2;
    printf("Asking server to add %d and %d: ",
           operand1, operand2);
    code = RXDEMO_Add(rxConnP, operand1, operand2, &sum);
    if (code) {
        printf("\n** Error in the RXDEMO_Add RPC: code is %d\n",
               code);
        exit(1);
    }
    printf("Reported sum is %d\n", sum);
```

The first argument to *RXDEMO_Add()* is a pointer to the *Rx* connection established above. The client-side body of the *RXDEMO_Add()* function was generated from the *rxdemo.xg* interface file, and resides in the *rxdemo.cs.c* file (see Section 6.3.1). It gives the appearance of being a normal C procedure call.

The second RPC invocation involves the more complex, streamed *RXDEMO_GetFile()* function. More of the internal *Rx* workings are exposed in this type of call. The first additional detail to consider is that we must manually create a new *Rx* call on the connection.

```
/*
 * Set up for our second, streamed procedure call.
 */
printf("Name of file to read from server: ");
scanf("%s", fileName);
maxBytesToRead = RXDEMO_BUFF_BYTES;
printf("Setting up an Rx call for RXDEMO_GetFile...");
rxCallP = rx_NewCall(rxConnP);
if (rxCallP == (struct rx_call *)0) {
    printf("** Can't create call\n");
    exit(1);
}
printf("done\n");
```

Once the *Rx* call structure has been created, we may begin executing the call itself. Having been declared to be `split` in the interface file, *Rxgen* creates *two* function bodies for *rxdemo_GetFile()* and places them in *rxdemo.cs.c*. The first, *StartRXDEMO_GetFile()*, is responsible for marshalling the outgoing arguments and issuing the RPC. The second, *EndRXDEMO_GetFile()*, takes care of unmarshalling the non-streamed `OUT` function parameters. The following code fragment illustrates how the RPC is started, using the *StartRXDEMO_GetFile()* routine to pass the call parameters to the server.

```
/*
 * Sending IN parameters for the streamed call.
 */
code = StartRXDEMO_GetFile(rxCallP, fileName);
if (code) {
    printf("** Error calling StartRXDEMO_GetFile(); code is %d\n",
           code);
    exit(1);
}
```

Once the call parameters have been shipped, the server will commence delivering the "stream" data bytes back to the client on the given *Rx* call structure. The first longword to come back on the stream specifies the number of bytes to follow.

```
/*
 * Begin reading the data being shipped from the server in response to
 * our setup call.  The first longword coming back on the Rx call is
 * the number of bytes to follow.  It appears in network byte order,
 * so we have to fix it up before referring to it.
 */
bytesReallyRead = rx_Read(rxCallP, &fileDataBytes, sizeof(long));
if (bytesReallyRead != sizeof(long)) {
    printf("** Only %d bytes read for file length; should have been %d\n",
           bytesReallyRead, sizeof(long));
    exit(1);
}
fileDataBytes = ntohl(fileDataBytes);
```

Once the client knows how many bytes will be sent, it runs a loop in which it reads a buffer at a time from the *Rx* call stream, using *rx_Read()* to accomplish this. In this application, all that is done with each newly-acquired buffer of information is printing it out.

```
/*
 * Read the file bytes via the Rx call, a buffer at a time.
 */
printf("[File contents (%d bytes) fetched over the Rx call appear below]\n\n",
       fileDataBytes);
while (fileDataBytes > 0) {
    currBytesToRead = (fileDataBytes > maxBytesToRead ?
                       maxBytesToRead : fileDataBytes);
    bytesReallyRead = rx_Read(rxCallP, buff, currBytesToRead);
    if (bytesReallyRead != currBytesToRead) {
        printf("\nExpecting %d bytes on this read, got %d instead\n",
               currBytesToRead, bytesReallyRead);
        exit(1);
    }
    /*
     * Null-terminate the chunk before printing it.
     */
    buff[currBytesToRead] = 0;
    printf("%s", buff);

    /*
     * Adjust the number of bytes left to read.
     */
    fileDataBytes -= currBytesToRead;

} /*Read one bufferful of the file*/
```

After this loop terminates, the *Rx* stream has been drained of all data. The *Rx* call is concluded by invoking the second of the two automatically-generated functions, *EndRXDEMO_GetFile()*, which retrieves the call's `OUT` parameter from the server.

```
/*
 * Finish off the Rx call, getting the OUT parameters.
 */
printf("\n\n[End of file data]\n");
code = EndRXDEMO_GetFile(rxCallP, &getResults);
if (code) {
    printf("** Error getting file transfer results; code is %d\n",
           code);
    exit(1);
}
```

With both normal and streamed *Rx* calls accomplished, the client demo code concludes by terminating the *Rx* call it set up earlier. With that done, the client exits.

```
/*
 * Finish off the Rx call.
 */
code = rx_EndCall(rxCallP, code);
if (code)
    printf("Error in calling rx_EndCall(); code is %d\n",
           code);

printf("\n\nrxdemo complete.\n");
```

6.2.3 Server Program: *rxdemo_server.c*

The *rxdemo* server program, *rxdemo_server*, implements the operations promised in the *rxdemo.xg* interface file.

After the initial header, the external function *RXDEMO_ExecuteRequest()* is declared. The *RXDEMO_ExecuteRequest()* function is generated automatically by *rxgen* from the interface file and deposited in *rxdemo.ss.c*. The main program listed below will associate this *RXDEMO_ExecuteRequest()* routine with the *Rx* service to be instantiated.

```
/*======================================================================%
% Server portion of the example RXDEMO application, using both         %
% standard and streamed calls.                                         %
%                                                                      %
% Edward R. Zayas                                                      %
% Transarc Corporation                                                 %
%                                                                      %
%                                                                      %
% The United States Government has rights in this work pursuant        %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
```

```
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                      %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                      %
% Redistribution and use in source and binary forms are permitted      %
% provided that: (1) source distributions retain this entire copy-     %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                      %
%       ``This product includes software developed by Transarc         %
%       Corporation and its contributors''                             %
%                                                                      %
% in the documentation or other materials mentioning features or       %
% use of this software.  Neither the name of Transarc nor the names    %
% of its contributors may be used to endorse or promote products       %
% derived from this software without specific prior written            %
% permission.                                                          %
%                                                                      %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                %
%=====================================================================*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <netdb.h>
#include <stdio.h>
#include "rxdemo.h"

#define N_SECURITY_OBJECTS 1

extern RXDEMO_ExecuteRequest();
```

After choosing either the default or user-specified UDP port on which the *Rx* service will be established, *rx_Init()* is called to set up the library.

```
main(argc, argv)
    int argc;
    char **argv;

{ /*Main*/

    static char pn[] = "rxdemo_server"; /*Program name*/
    struct rx_securityClass
        *(securityObjects[1]);          /*Security objs*/
    struct rx_service *rxServiceP;      /*Ptr to Rx service descriptor*/
    struct rx_call *rxCallP;            /*Ptr to Rx call descriptor*/
    int demoUDPPort;                    /*UDP port of Rx service*/
    int fd;                             /*File descriptor*/
    int code;                           /*Return code*/
```

```
    printf("\n%s: Example Rx server process\n\n", pn);
    if (argc > 2) {
        printf("Usage: rxdemo [PortToUse]");
        exit(1);
    }

    if (argc > 1)
        demoUDPPort = atoi(argv[1]);
    else
        demoUDPPort = RXDEMO_SERVER_PORT;

    /*
     * Initialize the Rx facility, telling it the UDP port number this
     * server will use for its single service.
     */
    printf("Listening on UDP port %d\n",
           demoUDPPort);
    code = rx_Init(demoUDPPort);
    if (code) {
        printf("** Error calling rx_Init(); code is %d\n",
               code);
        exit(1);
    }
```

A security object specific to the server side of an *Rx* conversation is created in the next code fragment. As with the client side of the code, a "null" server security object, namely one that does not perform any authentication at all, is constructed with the *rxnull_NewServerSecurityObject()* function.

```
    /*
     * Create a single server-side security object.  In this case, the
     * null security object (for unauthenticated connections) will be used
     * to control security on connections made to this server.
     */
    securityObjects[RXDEMO_NULL_SECOBJ_IDX] = rxnull_NewServerSecurityObject();
    if (securityObjects[RXDEMO_NULL_SECOBJ_IDX] == (struct rx_securityClass *) 0) {
        printf("** Can't create server-side security object\n");
        exit(1);
    }
```

The *rxdemo_server* program is now in a position to create the desired *Rx* service, primed to recognize exactly those interface calls defined in *rxdemo.xg*. This is accomplished by calling the *rx_NewService()* library routine, passing it the security object created above and the generated *Rx* dispatcher routine.

```
    /*
     * Instantiate a single sample service.  The rxgen-generated procedure
```

```
     * called to dispatch requests is passed in (RXDEMO_ExecuteRequest).
     */
    rxServiceP = rx_NewService(0,
                               RXDEMO_SERVICE_ID,
                               "rxdemo",
                               securityObjects,
                               1,
                               RXDEMO_ExecuteRequest);
    if (rxServiceP == (struct rx_service *) 0) {
        printf("** Can't create Rx service\n");
        exit(1);
    }
```

The final step in this main routine is to activate servicing of calls to the exported *Rx* interface. Specifically, the proper number of threads are created to handle incoming interface calls. Since we are passing a non-zero argument to the *rx_StartServer()* call, the main program will itself begin executing the server thread loop, never returning from the *rx_StartServer()* call. The print statement afterwards should never be executed, and its presence represents some level of paranoia, useful for debugging malfunctioning thread packages.

```
    /*
     * Start up Rx services, donating this thread to the server pool.
     */
    rx_StartServer(1);

    /*
     * We should never return from the previous call.
     */
    printf("** rx_StartServer() returned!!\n");
    exit(1);

}  /*Main*/
```

Following the main procedure are the functions called by the automatically-generated routines in the *rxdemo.ss.c* module to implement the specific routines defined in the *Rx* interface.

The first to be defined is the *RXDEMO_Add()* function. The arguments for this routine are exactly as they appear in the interface definition, with the exception of the very first. The `a_rxCallP` parameter is a pointer to the *Rx* structure describing the call on which this function was activated. All user-supplied routines implementing an interface function are required to have a pointer to this structure as their first parameter. Other than printing out the fact that it has been called and which operands it received, all that *RXDEMO_Add()* does is compute the sum and place it in the output parameter. Since

RXDEMO_Add() is a non-streamed function, with all data travelling through the set of parameters, this is all that needs to be done. To mark a successful completion, *RXDEMO_Add()* returns zero, which is passed all the way through to the RPC's client.

```
int RXDEMO_Add(a_rxCallP, a_operand1, a_operand2, a_resultP)
    struct rx_call *a_rxCallP;
    int a_operand1, a_operand2;
    int *a_resultP;

{ /*RXDEMO_Add*/

    printf("\t[Handling call to RXDEMO_Add(%d, %d)]\n",
           a_operand1, a_operand2);
    *a_resultP = a_operand1 + a_operand2;
    return(0);

}  /*RXDEMO_Add*/
```

The next and final interface routine defined in this file is *RXDEMO_GetFile()*. Declared as a `split` function in the interface file, *RXDEMO_GetFile()* is an example of a streamed *Rx* call. As with *RXDEMO_Add()*, the initial parameter is required to be a pointer to the *Rx* call structure with which this routine is associated, Similarly, the other parameters appear exactly as in the interface definition, and are handled identically.

The difference between *RXDEMO_Add()* and *RXDEMO_GetFile()* is in the use of the *rx_Write()* library routine by *RXDEMO_GetFile()* to feed the desired file's data directly into the *Rx* call stream. This is an example of the use of the *a_rxCallP* argument, providing all the information necessary to support the *rx_Write()* activity.

The *RXDEMO_GetFile()* function begins by printing out the fact that it's been called and the name of the requested file. It will then attempt to open the requested file and stat it to determine its size.

```
int RXDEMO_GetFile(a_rxCallP, a_nameToRead, a_resultP)
    struct rx_call *a_rxCallP;
    char *a_nameToRead;
    int *a_resultP;

{ /*RXDEMO_GetFile*/

    struct stat fileStat;           /*Stat structure for file*/
    long fileBytes;                 /*Size of file in bytes*/
    long nboFileBytes;              /*File bytes in network byte order*/
    int code;                       /*Return code*/
    int bytesReallyWritten;         /*Bytes written on Rx channel*/
    int bytesToSend;                /*Num bytes to read & send this time*/
    int maxBytesToSend;             /*Max num bytes to read & send ever*/
```

```
int bytesRead;                  /*Num bytes read from file*/
char buff[RXDEMO_BUFF_BYTES+1]; /*Read buffer*/
int fd;                         /*File descriptor*/

maxBytesToSend = RXDEMO_BUFF_BYTES;
printf("\t[Handling call to RXDEMO_GetFile(%s)]\n",
       a_nameToRead);
fd = open(a_nameToRead, O_RDONLY, 0444);
if (fd < 0) {
    printf("\t\t[**Can't open file '%s']\n",
                a_nameToRead);
    *a_resultP = RXDEMO_CODE_CANT_OPEN;
    return(1);
}
else
    printf("\t\t[File opened]\n");

/*
 * Stat the file to find out how big it is.
 */
code = fstat(fd, &fileStat);
if (code) {
    *a_resultP = RXDEMO_CODE_CANT_STAT;
    printf("\t\t[File closed]\n");
    close(fd);
    return(1);
}
fileBytes = fileStat.st_size;
printf("\t\t[File has %d bytes]\n", fileBytes);
```

Only standard UNIX operations have been used so far. Now that the file is open, we must first feed the size of the file, in bytes, to the *Rx* call stream. With this information, the client code can then determine how many bytes will follow on the stream. As with all data that flows through an *Rx* stream, the longword containing the file size, in bytes, must be converted to network byte order before being sent. This insures that the recipient may properly interpret the streamed information, regardless of its memory architecture.

```
nboFileBytes = htonl(fileBytes);

/*
 * Write out the size of the file to the Rx call.
 */
bytesReallyWritten = rx_Write(a_rxCallP, &nboFileBytes, sizeof(long));
if (bytesReallyWritten != sizeof(long)) {
    printf("** %d bytes written instead of %d for file length\n",
           bytesReallyWritten, sizeof(long));
    *a_resultP = RXDEMO_CODE_WRITE_ERROR;
    printf("\t\t[File closed]\n");
    close(fd);
    return(1);
}
```

Once the number of file bytes has been placed in the stream, the *RXDEMO_GetFile()* routine runs a loop, reading a buffer's worth of the file and then inserting that buffer of file data into the *Rx* stream at each iteration. This loop executes until all of the file's bytes have been shipped. Notice there is **no** special end-of-file character or marker inserted into the stream.

The body of the loop checks for both UNIX *read()* and *rx_Write* errors. If there is a problem reading from the UNIX file into the transfer buffer, it is reflected back to the client by setting the error return parameter appropriately. Specifically, an individual UNIX *read()* operation could fail to return the desired number of bytes. Problems with *rx_Write()* are handled similarly. All errors discovered in the loop result in the file being closed, and *RXDEMO_GetFile()* exiting with a non-zero return value.

```
/*
 * Write out the contents of the file, one buffer at a time.
 */
while (fileBytes > 0) {
    /*
     * Figure out the number of bytes to read (and send) this time.
     */
    bytesToSend = (fileBytes > maxBytesToSend ?
                   maxBytesToSend : fileBytes);
    bytesRead = read(fd, buff, bytesToSend);
    if (bytesRead != bytesToSend) {
        printf("Read %d instead of %d bytes from the file\n",
               bytesRead, bytesToSend);
        *a_resultP = RXDEMO_CODE_WRITE_ERROR;
        printf("\t\t[File closed]\n");
        close(fd);
        return(1);
    }

    /*
     * Go ahead and send them.
     */
    bytesReallyWritten = rx_Write(a_rxCallP, buff, bytesToSend);
    if (bytesReallyWritten != bytesToSend) {
        printf("%d file bytes written instead of %d\n",
               bytesReallyWritten, bytesToSend);
        *a_resultP = RXDEMO_CODE_WRITE_ERROR;
        printf("\t\t[File closed]\n");
        close(fd);
        return(1);
    }

    /*
     * Update the number of bytes left to go.
     */
    fileBytes -= bytesToSend;

} /*Write out the file to our caller*/
```

Once all of the file's bytes have been shipped to the remote client, all that remains to be done is to close the file and return successfully.

```
    /*
     * Close the file, then return happily.
     */
    *a_resultP = RXDEMO_CODE_SUCCESS;
    printf("\t\t[File closed]\n");
    close(fd);
    return(0);

}  /*RXDEMO_GetFile*/
```

6.2.4 Makefile

This file directs the compilation and installation of the *rxdemo* code. It specifies the locations of libraries, include files, sources, and such tools as *Rxgen* and `install`, which strips symbol tables from executables and places them in their target directories. This *Makefile* demostrates cross-cell software development, with the *rxdemo* sources residing in the `grand.central.org` cell and the AFS include files and libraries accessed from their locations in the `transarc.com` cell.

In order to produce and install the *rxdemo_server* and *rxdemo_client* binaries, the `system` target should be specified on the command line when invoking `make`:

```
make system
```

A note of caution is in order concerning generation of the *rxdemo* binaries. While tools exist that deposit the results of all compilations to other (architecture-specific) directories, and thus facilitate multiple simultaneous builds across a variety of machine architectures (e.g., Transarc's `washtool`), the assumption is made here that compilations will take place directly in the directory containing all the *rxdemo* sources. Thus, a user will have to execute a `make clean` command to remove all machine-specific object, library, and executable files before compiling for a different architecture. Note, though, that the binaries are installed into a directory specifically reserved for the current machine type. Specifically, the final pathname component of the `${PROJ_DIR}bin` installation target is really a symbolic link to `${PROJ_DIR}.bin/@sys`.

Two libraries are needed to support the *rxdemo* code. The first is obvious, namely the *Rx* `librx.a` library. The second is the lightweight thread package library, `liblwp.a`, which implements all the threading operations that must be performed. The include files are taken from the UNIX */usr/include* directory, along with various AFS-specific directories. Note that for portability reasons, this *Makefile* only contains fully-qualified AFS pathnames and "standard" UNIX pathnames (such as */usr/include*).

```
#=======================================================================#
# The United States Government has rights in this work pursuant         #
# to contract no. MDA972-90-C-0036 between the United States Defense    #
# Advanced Research Projects Agency and Transarc Corporation.           #
#                                                                       #
# (C) Copyright 1991 Transarc Corporation                               #
#                                                                       #
# Redistribution and use in source and binary forms are permitted       #
# provided that: (1) source distributions retain this entire copy-      #
# right notice and comment, and (2) distributions including binaries    #
# display the following acknowledgement:                                #
#                                                                       #
#       ``This product includes software developed by Transarc          #
#       Corporation and its contributors''                              #
#                                                                       #
# in the documentation or other materials mentioning features or        #
# use of this software.  Neither the name of Transarc nor the names     #
# of its contributors may be used to endorse or promote products        #
# derived from this software without specific prior written             #
# permission.                                                           #
#                                                                       #
# THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  #
# WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  #
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                 #
#=======================================================================#

SHELL           = /bin/sh
TOOL_CELL       = grand.central.org
AFS_INCLIB_CELL = transarc.com
USR_CONTRIB     = /afs/${TOOL_CELL}/darpa/usr/contrib/
PROJ_DIR        = ${USR_CONTRIB}.site/grand.central.org/rxdemo/
AFS_INCLIB_DIR  = /afs/${AFS_INCLIB_CELL}/afs/dest/
RXGEN           = ${AFS_INCLIB_DIR}bin/rxgen
INSTALL         = ${AFS_INCLIB_DIR}bin/install

LIBS    = ${AFS_INCLIB_DIR}lib/librx.a \
          ${AFS_INCLIB_DIR}lib/liblwp.a

CFLAGS = -g \
         -I. \
         -I${AFS_INCLIB_DIR}include \
         -I${AFS_INCLIB_DIR}include/afs \
         -I${AFS_INCLIB_DIR} \
         -I/usr/include

system: install

install: all
```

Example Server and Client

```
        ${INSTALL} rxdemo_client ${PROJ_DIR}bin
        ${INSTALL} rxdemo_server ${PROJ_DIR}bin

all: rxdemo_client rxdemo_server

rxdemo_client: rxdemo_client.o ${LIBS} rxdemo.cs.o
        ${CC} ${CFLAGS} -o rxdemo_client rxdemo_client.o rxdemo.cs.o ${LIBS}

rxdemo_server: rxdemo_server.o rxdemo.ss.o ${LIBS}
        ${CC} ${CFLAGS} -o rxdemo_server rxdemo_server.o rxdemo.ss.o ${LIBS}

rxdemo_client.o: rxdemo.h

rxdemo_server.o: rxdemo.h

rxdemo.cs.c rxdemo.ss.c rxdemo.er.c rxdemo.h: rxdemo.xg
        rxgen rxdemo.xg

clean:
        rm -f *.o rxdemo.cs.c rxdemo.ss.c rxdemo.xdr.c rxdemo.h \
                rxdemo_client rxdemo_server core
```

6.3 Computer-Generated Files

The four human-generated files described above provide all the information necessary to construct the full set of modules to support the *rxdemo* example application. This section describes those routines that are generated from the base set by *Rxgen*, filling out the code required to implement an *Rx* service.

6.3.1 Client-Side Routines: *rxdemo.cs.c*

The *rxdemo_client.c* program, described in Section 6.2.2, calls the client-side stub routines contained in this module in order to make *rxdemo* RPCs. Basically, these client-side stubs are responsible for creating new *Rx* calls on the given connection parameter and then marshalling and unmarshalling the rest of the interface call parameters. The `IN` and `INOUT` arguments, namely those that are to be delivered to the server-side code implementing the call, must be packaged in network byte order and shipped along the given *Rx* call. The return parameters, namely those objects declared as `INOUT` and `OUT`, must be fetched from the server side of the associated *Rx* call, put back in host byte order, and inserted into the appropriate parameter variables.

The first part of *rxdemo.cs.c* echoes the definitions appearing in the *rxdemo.xg* interface file, and also `#include`s another *Rxgen*-generated file, *rxdemo.h*.

```
/*======================================================================%
% Edward R. Zayas                                                       %
% Transarc Corporation                                                  %
%                                                                       %
%                                                                       %
% The United States Government has rights in this work pursuant         %
% to contract no. MDA972-90-C-0036 between the United States Defense    %
% Advanced Research Projects Agency and Transarc Corporation.           %
%                                                                       %
% (C) Copyright 1991 Transarc Corporation                               %
%                                                                       %
% Redistribution and use in source and binary forms are permitted       %
% provided that: (1) source distributions retain this entire copy-      %
% right notice and comment, and (2) distributions including binaries    %
% display the following acknowledgement:                                %
%                                                                       %
%       ``This product includes software developed by Transarc          %
%       Corporation and its contributors''                              %
%                                                                       %
% in the documentation or other materials mentioning features or        %
% use of this software.  Neither the name of Transarc nor the names     %
% of its contributors may be used to endorse or promote products        %
% derived from this software without specific prior written             %
% permission.                                                           %
%                                                                       %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                 %
%======================================================================*/

/* Machine generated file -- Do NOT edit */

#include "rxdemo.h"

#include <rx/rx.h>
#include <rx/rx_null.h>
#define RXDEMO_SERVER_PORT      8000    /*Service port to advertise*/
#define RXDEMO_SERVICE_PORT        0    /*User server's port*/
#define RXDEMO_SERVICE_ID          4    /*Service ID*/
#define RXDEMO_NULL_SECOBJ_IDX     0    /*Index of null security object*/
#define RXDEMO_MAX      3
#define RXDEMO_MIN      2
#define RXDEMO_NULL     0
#define RXDEMO_NAME_MAX_CHARS   64
#define RXDEMO_BUFF_BYTES 512
#define RXDEMO_CODE_SUCCESS       0
#define RXDEMO_CODE_CANT_OPEN     1
#define RXDEMO_CODE_CANT_STAT     2
#define RXDEMO_CODE_CANT_READ     3
#define RXDEMO_CODE_WRITE_ERROR 4
```

The next code fragment defines the client-side stub for the *RXDEMO_Add()* routine, called by the *rxdemo_client* program to execute the associated RPC.

```
int RXDEMO_Add(z_conn, a, b, result)
        register struct rx_connection *z_conn;
        int a, b;
        int * result;
{
        struct rx_call *z_call = rx_NewCall(z_conn);
        static int z_op = 1;
        int z_result;
        XDR z_xdrs;

        xdrrx_create(&z_xdrs, z_call, XDR_ENCODE);

        /* Marshal the arguments */
        if ((!xdr_int(&z_xdrs, &z_op))
             || (!xdr_int(&z_xdrs, &a))
             || (!xdr_int(&z_xdrs, &b))) {
                z_result = RXGEN_CC_MARSHAL;
                goto fail;
        }

        /* Un-marshal the reply arguments */
        z_xdrs.x_op = XDR_DECODE;
        if ((!xdr_int(&z_xdrs, result))) {
                z_result = RXGEN_CC_UNMARSHAL;
                goto fail;
        }

        z_result = RXGEN_SUCCESS;
fail:
        return rx_EndCall(z_call, z_result);
}
```

The very first operation performed by *RXDEMO_Add()* occurs in the local variable declarations, where `z_call` is set to point to the structure describing a newly-created *Rx* call on the given connection. An XDR structure, `z_xdrs`, is then created for the given *Rx* call with *xdrrx_create()*. This XDR object is used to deliver the proper arguments, in network byte order, to the matching server stub code. Three calls to *xdr_int()* follow, which insert the appropriate *Rx* opcode and the two operands into the *Rx* call. With the *IN* arguments thus transmitted, *RXDEMO_Add()* prepares to pull the value of the single `OUT` parameter. The `z_xdrs` XDR structure, originally set to `XDR_ENCODE` objects, is now reset to `XDR_DECODE` to convert further items received into host byte order. Once the return parameter promised by the function is retrieved, *RXDEMO_Add()* returns successfully.

Should any failure occur in passing the parameters to and from the server side of the call, the branch to `fail` will invoke *Rx*_EndCall(), which advises the server that the call has come to a premature end (see Section 5.6.6 for full details on *rx_EndCall()* and the meaning of its return value).

The next client-side stub appearing in this generated file handles the delivery of the IN parameters for *StartRXDEMO_GetFile()*. It operates identically as the *RXDEMO_Add()* stub routine in this respect, except that it does not attempt to retrieve the OUT parameter. Since this is a streamed call, the number of bytes that will be placed on the *Rx* stream cannot be determined at compile time, and must be handled explicitly by *rxdemo_client.c*.

```
int StartRXDEMO_GetFile(z_call, a_nameToRead)
        register struct rx_call *z_call;
        char * a_nameToRead;
{
        static int z_op = 2;
        int z_result;
        XDR z_xdrs;

        xdrrx_create(&z_xdrs, z_call, XDR_ENCODE);

        /* Marshal the arguments */
        if ((!xdr_int(&z_xdrs, &z_op))
             || (!xdr_string(&z_xdrs, &a_nameToRead, RXDEMO_NAME_MAX_CHARS))) {
                z_result = RXGEN_CC_MARSHAL;
                goto fail;
        }

        z_result = RXGEN_SUCCESS;
fail:
        return z_result;
}
```

The final stub routine appearing in this generated file, `EndRXDEMO_GetFile()`, handles the case where *rxdemo_client.c* has already successfully recovered the unbounded streamed data appearing on the call, and then simply has to fetch the OUT parameter. This routine behaves identially to the latter portion of *RXDEMO_GetFile()*.

```
int EndRXDEMO_GetFile(z_call, a_result)
        register struct rx_call *z_call;
        int * a_result;
{
        int z_result;
        XDR z_xdrs;

        /* Un-marshal the reply arguments */
        xdrrx_create(&z_xdrs, z_call, XDR_DECODE);
        if ((!xdr_int(&z_xdrs, a_result))) {
                z_result = RXGEN_CC_UNMARSHAL;
                goto fail;
        }
```

```
        z_result = RXGEN_SUCCESS;
fail:
        return z_result;
}
```

6.3.2 Server-Side Routines: *rxdemo.ss.c*

This generated file provides the core components required to implement the server side of the *rxdemo* RPC service. Included in this file is the generated dispatcher routine, *RXDEMO_ExecuteRequest()*, which the *rx_NewService()* invocation in *rxdemo_server.c* uses to construct the body of each listener thread's loop. Also included are the server-side stubs to handle marshalling and unmarshalling of parameters for each defined RPC call (i.e., *_RXDEMO_Add()* and *_RXDEMO_GetFile()*). These stubs are called by *RXDEMO_ExecuteRequest()*. The routine to be called by *RXDEMO_ExecuteRequest()* depends on the opcode received, which appears as the very first longword in the call data.

As usual, the first fragment is copyright information followed by the body of the definitions from the interface file.

```
/*======================================================================%
% Edward R. Zayas                                                       %
% Transarc Corporation                                                  %
%                                                                       %
%                                                                       %
% The United States Government has rights in this work pursuant        %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                       %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                       %
% Redistribution and use in source and binary forms are permitted      %
% provided that: (1) source distributions retain this entire copy-     %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                       %
%       ``This product includes software developed by Transarc         %
%       Corporation and its contributors''                             %
%                                                                       %
% in the documentation or other materials mentioning features or       %
% use of this software.  Neither the name of Transarc nor the names    %
% of its contributors may be used to endorse or promote products       %
% derived from this software without specific prior written            %
% permission.                                                           %
%                                                                       %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
```

```
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                    %
%====================================================================*/

/* Machine generated file -- Do NOT edit */

#include "rxdemo.h"

#include <rx/rx.h>
#include <rx/rx_null.h>
#define RXDEMO_SERVER_PORT      8000    /*Service port to advertise*/
#define RXDEMO_SERVICE_PORT        0    /*User server's port*/
#define RXDEMO_SERVICE_ID          4    /*Service ID*/
#define RXDEMO_NULL_SECOBJ_IDX     0    /*Index of null security object*/
#define RXDEMO_MAX      3
#define RXDEMO_MIN      2
#define RXDEMO_NULL     0
#define RXDEMO_NAME_MAX_CHARS   64
#define RXDEMO_BUFF_BYTES 512
#define RXDEMO_CODE_SUCCESS      0
#define RXDEMO_CODE_CANT_OPEN    1
#define RXDEMO_CODE_CANT_STAT    2
#define RXDEMO_CODE_CANT_READ    3
#define RXDEMO_CODE_WRITE_ERROR 4
```

After this preamble, the first server-side stub appears. This *_RXDEMO_Add()* routine is basically the inverse of the *RXDEMO_Add()* client-side stub defined in *rxdemo.cs.c*. Its job is to unmarshall the *IN* parameters for the call, invoke the "true" server-side *RXDEMO_Add()* routine (defined in *rxdemo_server.c*), and then package and ship the `OUT` parameter. Being so similar to the client-side *RXDEMO_Add()*, no further discussion is offered here.

```
long _RXDEMO_Add(z_call, z_xdrs)
        struct rx_call *z_call;
        XDR *z_xdrs;
{
        long z_result;
        int a, b;
        int result;

        if ((!xdr_int(z_xdrs, &a))
             || (!xdr_int(z_xdrs, &b))) {
                z_result = RXGEN_SS_UNMARSHAL;
                goto fail;
        }

        z_result = RXDEMO_Add(z_call, a, b, &result);
        z_xdrs->x_op = XDR_ENCODE;
        if ((!xdr_int(z_xdrs, &result)))
                z_result = RXGEN_SS_MARSHAL;
fail:
```

```
        return z_result;
}
```

The second server-side stub, *_RXDEMO_GetFile()*, appears next. It operates identically to *_RXDEMO_Add()*, first unmarshalling the `IN` arguments, then invoking the routine that actually performs the server-side work for the call, then finishing up by returning the `OUT` parameters.

```
long _RXDEMO_GetFile(z_call, z_xdrs)
        struct rx_call *z_call;
        XDR *z_xdrs;
{
        long z_result;
        char * a_nameToRead=(char *)0;
        int a_result;

        if ((!xdr_string(z_xdrs, &a_nameToRead, RXDEMO_NAME_MAX_CHARS))) {
                z_result = RXGEN_SS_UNMARSHAL;
                goto fail;
        }

        z_result = RXDEMO_GetFile(z_call, a_nameToRead, &a_result);
        z_xdrs->x_op = XDR_ENCODE;
        if ((!xdr_int(z_xdrs, &a_result)))
                z_result = RXGEN_SS_MARSHAL;
fail:
        z_xdrs->x_op = XDR_FREE;
        if (!xdr_string(z_xdrs, &a_nameToRead, RXDEMO_NAME_MAX_CHARS)) goto fail1;
        return z_result;
fail1:
        return RXGEN_SS_XDRFREE;
}
```

The next portion of the automatically generated server-side module sets up the dispatcher routine for incoming *Rx* calls. The above stub routines are placed into an array in opcode order.

```
long _RXDEMO_Add();
long _RXDEMO_GetFile();

static long (*StubProcsArray0[])() = {_RXDEMO_Add, _RXDEMO_GetFile};
```

The dispatcher routine itself, *RXDEMO_ExecuteRequest*, appears next. This is the function provided to the *rx_NewService()* call in *rxdemo_server.c*, and it is used as the body of each listener thread's service loop. When activated, it decodes the first longword in the given *Rx* call, which contains the opcode. It then dispatches the call based on this opcode, invoking the appropriate server-side stub as organized in the `StubProcsArray`.

```
RXDEMO_ExecuteRequest(z_call)
        register struct rx_call *z_call;
{
        int op;
        XDR z_xdrs;
        long z_result;

        xdrrx_create(&z_xdrs, z_call, XDR_DECODE);
        if (!xdr_int(&z_xdrs, &op))
                z_result = RXGEN_DECODE;
        else if (op < RXDEMO_LOWEST_OPCODE || op > RXDEMO_HIGHEST_OPCODE)
                z_result = RXGEN_OPCODE;
        else
                z_result = (*StubProcsArray0[op - RXDEMO_LOWEST_OPCODE])(z_call, &z_xdrs);
        return z_result;
}
```

6.3.3 External Data Rep File: *rxdemo.xdr.c*

This file is created to provide the special routines needed to map any user-defined structures appearing as *Rx* arguments into and out of network byte order. Again, all on-the-wire data appears in network byte order, insuring proper communication between servers and clients with different memory organizations.

Since the *rxdemo* example application does not define any special structures to pass as arguments in its calls, this generated file contains only the set of definitions appearing in the interface file. In general, though, should the user define a `struct xyz` and use it as a parameter to an RPC function, this file would contain a routine named *xdr_xyz()*, which converted the structure field-by-field to and from network byte order.

```
/*=======================================================================%
% Edward R. Zayas                                                       %
% Transarc Corporation                                                  %
%                                                                       %
%                                                                       %
% The United States Government has rights in this work pursuant        %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                       %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                       %
% Redistribution and use in source and binary forms are permitted      %
% provided that: (1) source distributions retain this entire copy-     %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                       %
%       ``This product includes software developed by Transarc         %
```

```
%       Corporation and its contributors''                                 %
%                                                                          %
% in the documentation or other materials mentioning features or          %
% use of this software.  Neither the name of Transarc nor the names       %
% of its contributors may be used to endorse or promote products          %
% derived from this software without specific prior written               %
% permission.                                                              %
%                                                                          %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED    %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF    %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                   %
%=========================================================================*/

/* Machine generated file -- Do NOT edit */

#include "rxdemo.h"

#include <rx/rx.h>
#include <rx/rx_null.h>
#define RXDEMO_SERVER_PORT      8000    /*Service port to advertise*/
#define RXDEMO_SERVICE_PORT        0    /*User server's port*/
#define RXDEMO_SERVICE_ID          4    /*Service ID*/
#define RXDEMO_NULL_SECOBJ_IDX     0    /*Index of null security object*/
#define RXDEMO_MAX      3
#define RXDEMO_MIN      2
#define RXDEMO_NULL     0
#define RXDEMO_NAME_MAX_CHARS   64
#define RXDEMO_BUFF_BYTES 512
#define RXDEMO_CODE_SUCCESS      0
#define RXDEMO_CODE_CANT_OPEN    1
#define RXDEMO_CODE_CANT_STAT    2
#define RXDEMO_CODE_CANT_READ    3
#define RXDEMO_CODE_WRITE_ERROR 4
```

6.4 Sample Output

This section contains the output generated by running the example *rxdemo_server* and *rxdemo_client* programs described above. The server end was run on a machine named `Apollo`, and the client program was run on a machine named `Bigtime`.

The server program on `Apollo` was started as follows:

```
apollo: rxdemo_server

rxdemo_server: Example Rx server process

Listening on UDP port 8000
```

At this point, *rxdemo_server* has initialized its *Rx* module and started up its listener LWPs, which are sleeping on the arrival of an RPC from any *rxdemo* client.

The client portion was then started on `Bigtime`:

```
bigtime: rxdemo_client apollo

rxdemo: Example Rx client process

Connecting to Rx server on 'apollo', IP address 0x1acf37c0, UDP port 8000
  ---> Connected.
Asking server to add 1 and 2: Reported sum is 3
```

The command line instructs *rxdemo_client* to connect to the *rxdemo* server on host `apollo` and to use the standard port defined for this service. It reports on the successful *Rx* connection establishment, and immediately executes an *rxdemo_Add(1, 2)* RPC. It reports that the sum was successfully received. When the RPC request arrived at the server and was dispatched by the *rxdemo_server* code, it printed out the following line:

```
        [Handling call to RXDEMO_Add(1, 2)]
```

Next, *rxdemo_client* prompts for the name of the file to read from the *rxdemo* server. It is told to fetch the *Makefile* for the *Rx* demo directory. The server is executing in the same directory in which it was compiled, so an absolute name for the *Makefile* is not required. The client echoes the following:

```
Name of file to read from server: Makefile
Setting up an Rx call for RXDEMO_GetFile...done
```

As with the *rxdemo_Add()* call, *rxdemo_server* receives this RPC, and prints out the following information:

```
        [Handling call to RXDEMO_GetFile(Makefile)]
                [File opened]
                [File has 2450 bytes]
                [File closed]
```

It successfully opens the named file, and reports on its size in bytes. The *rxdemo_server* program then executes the streamed portion of the *rxdemo_GetFile* call, and when complete, indicates that the file has been closed. Meanwhile, *rxdemo_client* prints out the reported size of the file, follows it with the file's contents, then advises that the test run has completed:

```
[File contents (2450 bytes) fetched over the Rx call appear below]

#=======================================================================#
# The United States Government has rights in this work pursuant         #
# to contract no. MDA972-90-C-0036 between the United States Defense    #
# Advanced Research Projects Agency and Transarc Corporation.           #
#                                                                       #
# (C) Copyright 1991 Transarc Corporation                               #
#                                                                       #
# Redistribution and use in source and binary forms are permitted       #
# provided that: (1) source distributions retain this entire copy-      #
# right notice and comment, and (2) distributions including binaries    #
# display the following acknowledgement:                                #
#                                                                       #
#       ``This product includes software developed by Transarc          #
#       Corporation and its contributors''                              #
#                                                                       #
# in the documentation or other materials mentioning features or        #
# use of this software.  Neither the name of Transarc nor the names     #
# of its contributors may be used to endorse or promote products        #
# derived from this software without specific prior written             #
# permission.                                                           #
#                                                                       #
# THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  #
# WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  #
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                 #
#=======================================================================#

SHELL           = /bin/sh
TOOL_CELL       = grand.central.org
AFS_INCLIB_CELL = transarc.com
USR_CONTRIB     = /afs/${TOOL_CELL}/darpa/usr/contrib/
PROJ_DIR        = ${USR_CONTRIB}.site/grand.central.org/rxdemo/
AFS_INCLIB_DIR  = /afs/${AFS_INCLIB_CELL}/afs/dest/
RXGEN           = ${AFS_INCLIB_DIR}bin/rxgen
INSTALL         = ${AFS_INCLIB_DIR}bin/install

LIBS    = ${AFS_INCLIB_DIR}lib/librx.a \
          ${AFS_INCLIB_DIR}lib/liblwp.a

CFLAGS = -g \
         -I. \
         -I${AFS_INCLIB_DIR}include \
         -I${AFS_INCLIB_DIR}include/afs \
         -I${AFS_INCLIB_DIR} \
         -I/usr/include

system: install

install: all
        ${INSTALL} rxdemo_client ${PROJ_DIR}bin
        ${INSTALL} rxdemo_server ${PROJ_DIR}bin

all: rxdemo_client rxdemo_server
```

```
rxdemo_client: rxdemo_client.o ${LIBS} rxdemo.cs.o
        ${CC} ${CFLAGS} -o rxdemo_client rxdemo_client.o rxdemo.cs.o ${LIBS}

rxdemo_server: rxdemo_server.o rxdemo.ss.o ${LIBS}
        ${CC} ${CFLAGS} -o rxdemo_server rxdemo_server.o rxdemo.ss.o ${LIBS}

rxdemo_client.o: rxdemo.h

rxdemo_server.o: rxdemo.h

rxdemo.cs.c rxdemo.ss.c rxdemo.er.c rxdemo.h: rxdemo.xg
        rxgen rxdemo.xg

clean:
        rm -f *.o rxdemo.cs.c rxdemo.ss.c rxdemo.xdr.c rxdemo.h \
                rxdemo_client rxdemo_server core


[End of file data]


rxdemo complete.
```

The *rxdemo_server* program continues to run after handling these calls, offering its services to any other callers. It can be killed by sending it an interrupt signal using Control-C (or whatever mapping has been set up for the shell's interrupt character).

Bibliography

[1] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[2] S.P. Miller, B.C. Neuman, J.I. Schiller, J.H. Saltzer. *Kerberos Authentication and Authorization System*, Project Athena Technical Plan, Section E.2.1, M.I.T., December 1987.

[3] Bill Bryant. *Designing an Authentication System: a Dialogue in Four Scenes*, Project Athena internal document, M.I.T, draft of 8 February 1988.

[4] S. R. Kleinman. *Vnodes: An Architecture for Multiple file System Types in Sun UNIX*, Conference Proceedings, 1986 Summer Usenix Technical Conference, pp. 238-247, El Toro, CA, 1986.

Authentication Server Interface

Chapter 1

Overview

1.1 Introduction

The Authentication Server, or AuthServer, is a program that runs on one or more machines in a distributed workstation environment. In such an environment, it is common for a collection of server processes to provide facilities to the network community that are not available to individual workstations. In a large community, a formal mechanism to confidently identify clients to servers and vice versa is necessary. The AuthServer mediates this identification process. It provides information to the server that allows it to name its client. Conversely, the client can be confident it is in communication with the desired server and not with a Trojan Horse. This identification is not perfect but can be relied upon given certain assumptions about the security of the network, the integrity of the human administrators, and the cryptographic effort that might be expended to forge an identification.

1.2 Scope of this Document

This paper describes the design and structure of the AFS Authentication Server. The scope of this work is to provide readers with a sufficiently detailed description of the AuthServer so that they may construct client applications that call the server's RPC interface.

1.3 Document Layout

The second chapter discusses various aspects of the Auth Server's architecture. The basis of authentication in this system is a *shared secret* known by both parties in a communication. This secret is usually called a *key* or sometimes an *Encryption Key*. After this is discussed, the security issue is considered and assumptions made. Chapter three discusses the Auth Server's API in detail.

1.3.1 Related Documents

This document is a member of a documentation suite providing programmer level specifications for the operations of the various AFS servers and agents and the interfaces they export, as well as the underlying RPC system they use to communicate. The full suite of related AFS specification documents is listed below:

- *AFS-3 Programmer's Reference: Architectural Overview*: This paper provides an architectural overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and, examining their interactions.

- *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*: This document specifies the design and operation of the remote procedure call and lightweight process packages used by AFS.

- *AFS-3 Programmer's Reference: BOS Server Interface*: This paper describes the "nanny" service which assists in the administrability of the AFS environment.

- *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*: This document describes the workings and interfaces of the two primary AFS agents: the *File Server* and *Cache Manager*. The *File Server* provides a centralized disk repository for sets of files, regulating access to them. End users sitting on client machines rely on the *Cache Manager* agent, running in their kernel, to act as their agent in accessing the files stored on *File Server* machines, making those files appear as if they were really housed locally.

- *AFS-3 Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for providing two-way mappings between printable user names and their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users suitable for placement on ACLs.

- *AFS-3 Programmer's Reference: Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.

In addition to these papers, the AFS product is delivered with its own user, administrator, installation, and command reference documents.

Chapter 2

Authentication Server **Architecture**

This chapter presents the general architecture of the Authentication Server. First, mutual authentication is discussed in detail. Following this, some assumptions such as physical security and passwords are discussed. Finally, the security issue is discussed.

2.1 Encryption Key

2.1.1 Overview

The basis of authentication in this system is a *shared secret* known by both parties in a communication. This secret is usually called a *key* or sometimes an *Encryption Key* or *password*. A simple protocol can be used by each side to determine whether the other truly does know the correct key without actually sending the key across the network. The *Data Encryption Standard (DES)* is used to provide this determination, although other key-based encryption systems could be used. The AuthServer provides the client, directly, and the server, indirectly, with the shared secret which they use to identify each other.

2.1.2 Mutual Authentication

To deliver keys to the client and server, the AuthServer must use an untrustworthy communication channel. Because of this fundamental problem, each identity, including both clients and servers, must be registered with the AuthServer through secure means. Generally, this involves a face-to-face meeting with a new user and a system administrator to establish a user name and initial password. Such a registered user is called a principal identity or just a *principal*. The user's password is used to generate an encryption key which is used in all further transactions; the password itself is discarded. The user should change his password periodically and can do so without the intervention of a system administrator.

The key, generated from the user's password and stored in the AuthServer, is then used to *mutually authenticate* any client-server pair. The basic algorithm is quite simple. The client calls the AuthServer with the name of the server it wishes to contact and a random number which has been encrypted with the client's key. The AuthServer decrypts the packet with the client's key, which it knows, to obtain the random number. It then looks up the key of the server and creates a *ticket* encrypted with the server key and containing the client's name and a *session key*. Then the AuthServer creates a packet for return to the client which contains the random number, the session key, and the ticket prepared for the desired server all encrypted with the client's key. This packet contains two copies of the session key: one encrypted with the client's key and one encrypted two times: first with the server's key then with the client's key. This session key will be the secret shared by the client and server when they establish communications.

The client decrypts the packet returned from the AuthServer and checks the random number to see that it matches the one sent. If the number matches, it can be sure the AuthServer is genuine and really knows the client's key. The client saves the session key and delivers the ticket, which it can not interpret, to the server when the connection is opened. The server decrypts the ticket and verifies that the user name it contains is allowed to use its service. The session key can be used to encrypt communication between the two. The server knows who this user is because the name was in the ticket encrypted by the AuthServer. The client knows it is in communication with the real server since the server was able to obtain the session key from the the ticket created by the AuthServer. The parties can work in privacy, both confident of the identity of the other. This is *secure mutually authenticated communication*.

In practice a slightly more complicated system is employed for creating server tickets. The process of getting a ticket for a server is divided into two steps. The first involves contacting the AuthServer using the protocol outlined above. The AuthServer produces a ticket, called the AuthTicket, which is given to an intermediate server called the Ticket Granting Service (TGS). The second step is to contact the TGS, using the AuthTicket as proof of authentication, to obtain a ticket for the desired server. This second step may be repeated many times during a login session to get tickets for various servers as they are needed. The advantage of dividing the process this way is that the user's password is used only for the first step. Since the AuthTicket has a lifetime of a few hours, it is a smaller security risk to keep it in memory for the duration of a login session than to keep a password for the same period.

2.1.3 System Design

The design of this system borrows heavily from the work Needham and Schroeder[7] did on using encryption in an insecure network environment. Many ideas have been adapted from their work, especially the format of the tickets and much of their terminology.

A good deal of effort has gone into making the system compatible with Kerberos[6]. In this paper we refer to *cells* which are broadly equivalent to the *realms* of Kerberos. We have mostly adopted the name *ticket*, but earlier descriptions of the Andrew system referred to them as *tokens*. The term tokens is used here to refer to a structure containing a ticket and the associated session key.

2.2 Assumptions

2.2.1 Introduction

The security and reliability of the AuthServer depends on several assumptions about its environment. Making these assumptions explicit should help prevent security violations or other problems caused by inadvertently changing some aspect of the system on which the AuthServer relies. In this section we cover all the parts of the system that are unusual or critical to correct operation.

2.2.2 Physical Security

It is critical to the secure operation of the AuthServer that the computer on which it runs is a *physically secure machine*. There are many ways to compromise the integrity of the AuthServer if access to the hardware is possible. Many of these are difficult to combat or even detect. Providing physical security is an established technique with well known advantages and well understood weaknesses.

To obtain the many advantages of multiprocessing, the AuthServer is usually run as a distributed process on several machines. Each of these machines must be physically secure. The connection between the AuthServer's machines and the rest of the system, including both servers and clients, does not have to be secure. The protocols used to establish and maintain communication between the AuthServer and clients and between clients and servers attempt to protect against all types of attacks on the network. In addition, the physical security of neither the client nor other server machines is depended upon, as they may be owned and operated by individuals or groups throughout the user community.

2.2.3 Time Synchrony

The protocols depend upon a degree of *time synchrony* to detect stale or replayed messages which may be used in an attack on system security. Because perfect synchronization is unrealistic, this protocol assumes that clocks on different machines are within fifteen minutes of each other. This requirement is only important for changing passwords, because this is an operation with a side-effect. This synchronization limit means that a user who changes his or her password twice within fifteen minutes could have it changed back by an attacker replaying the first change request.

2.2.4 Passwords

The security of the system critically depends on the security of user passwords. Most users do not keep their password's safe or choose appropriate passwords. This is usually the largest source of security problems in any system. These problems, however, are limited to the compromise of individual users, although individuals with access to sensitive data need to be careful about their passwords.

The problem of managing server passwords is much more critical. The ability to reboot a server and have the server's password entered without human intervention is important for large systems that need to provide a high level of availability. To accomplish this, server passwords are stored on the local, physically secure disk. This approach is a reasonable interim solution but one which has many potential risks. The AuthServer simply assumes that all passwords are safe.

Another problem is the possibility of a *Trojan Horse* replacing the standard login program. Because fetching the login program over the network usually precedes the authentication process, there is no protection from having a rogue file server supply a copy of login that steals passwords. The same general problem applies to booting a workstation from a network server. These problems are not addressed by the AuthServer and are assumed to be controlled in other ways.

2.3 Security

2.3.1 Introduction

The basic authentication problem has two parts: establishing identity and defeating attempts to forge identities. The latter almost totally dictates the details of the solution. The job of establishing the identity of the user is simply a matter of determining whether or not a user knows the password associated with a principal. This could be as simple as having the user sit down at a terminal and type in his name and password which is then sent over the network to the AuthServer. To provide security in a potentially hostile environment, various counter-measures must be taken to prevent a user's password from being revealed and to prevent one user from masquerading as another. The basic source of security loopholes comes from the fact that both the local workstations and the network that connects them to the AuthServer are easily accessible to an attacker. However, these weaknesses are an intrinsic part of a distributed workstation environment so the design of the protocols takes them into account.

2.3.2 Security through Encryption

Security problems are primarily addressed by encrypting sensitive messages with DES. The choice of key and the details of the message contents vary according to the situation. The DES algorithm has several features that are important to its use in authentication [8].

- It has a relatively compact, eight byte key that can be formed from several convenient sources such as passwords or random numbers.
- Data of any length can be encrypted and decrypted given the key, but it is very difficult to do either without the key. The data must be padded to a multiple of eight bytes. The standard method of encrypting a sequence of data longer than eight bytes is called *cipher block chaining*. This method uses some data from the encryption of the previous eight byte chunk to encrypt the current chunk. This means that a change to a message will affect the translation of all subsequent data in the message.
- The key must be kept hidden.

The basic paradigm for achieving mutual authentication involves exchanging messages that can be recognized as correct when decrypted using the shared secret as the key. The properties of DES are such that without knowledge of the key it is not feasible to create a properly encrypted message or to decrypt a message. The choice of the message contents results from a consideration of the various methods of attack that the system must protect itself against.

2.3.3 Methods of Attack

The attacks that the design attempts to cope with fall into these categories:

- eavesdropping - This attack assumes that someone can listen in on a conversation and obtain private information. It includes both intercepting keys or passwords, which would be very serious and simple snooping on the contents of personal files.
- tampering - In this type of attack, data is modified, either in transit or by intercepting a packet, altered, and reinjected into the network.
- replay - This is reinsertion of old packets into the network in an attempt to deceive the recipient. The advantage of this approach is that the attacker does not need full knowledge of the contents of the packets. It is conceptually similar to tampering except that it assumes that the original packet was received or that the delay between interception and reinjection is relatively long.
- server misrepresentation - This is a specific case of a Trojan Horse where an attacker pretends to be a server. Mutual authentication is important in this case.
- cryptographic attack - This is an attempt to decrypt the encryption algorithm. Apart from using a encryption algorithm blessed by the National Security Agency (DES), little attempt is made to worry about this.

Eavesdropping implies that anything sent over the network can be read. This means that sensitive data must be encrypted when sent over the network. This primarily affects keys.

Encryptions can be used to detect tampering. The data to be protected must be encrypted even though the data itself may not be sensitive. If an encrypted message is tampered with, the decryption process will scramble it in ways impossible for the attacker to predict. With suitable consistency checks on the decrypted message, tampering can be detected and such messages rejected. By using encryption to prevent tampering, the recipient can assume that the data in a message is internally consistent.

Replay attacks can be defeated in one of two ways. Either by including a timestamp that causes old messages to be rejected or by including an identifier specified by the recipient in an earlier message. The advantage of timestamps is that they are easy to use; a single test by the server can detect a replayed message. The disadvantage is that the inevitable skew between the clocks of different machines and the routing delays imposed by the network puts a lower bound on the delay that can be detected by rejecting messages with "old" timestamps. Because many systems contain clock skews of several minutes, this test cannot be made very rigorous.

Using a handshaking algorithm that exchanges identifiers has the advantage that the identifiers are only used once. If the identifier is encountered in a replayed message after the original transaction has completed, the message will always be rejected. The problem is that the protocol for exchanging identifiers is more complex. The basic scenario is as follows:

client: $\{$r_1 = random()$\}^{key}$ $--\ >$

$<\ --$ $\{$r_1+1, r_2

= random()$\}^{key}$ **:server**

At this point the client knows that the server is real, but the server is still not sure about the client. Since the transaction is initiated by the client it gets the first information back. It takes one more message

client: $\{$r_2+1$\}^{key}$ $--\ >$

to convince the server and complete the mutual authentication.

An additional safeguard against replay is to test the source of each packet against the host the previous packet came from and make sure that all packets are from the same host. This makes hijacking an ongoing connection more difficult. There are two problems with using the host address:

- The hardware that inserts the host address may be altered to fake any desired return address.
- The user may intentionally change hosts from time to time which limits the strictness of this test to the lifetime of a single connection.

The problem of server misrepresentation is resolved whenever mutual authentication happens before sensitive data is exchanged. Alternatively, if sensitive data is encrypted, and assuming incorrect decryption can be detected with consistency checks, illegitimate servers can be rejected without authentication. In this case the encrypted data transfer accomplishes the authentication.

Protection from cryptographic attack is beyond the scope of this mechanism. It is implicitly assumed that anyone prepared to do a serious cryptographic attack on DES encrypted data in this system can have whatever he or she can get. Any data so sensitive that this might be a problem should be subject to additional safety measures. No such data should be stored in the Andrew File System.

Several simple design rules should serve to make such attacks more difficult. The goal of these rules is to not give away any more information than necessary.

- Keep messages short.
- Make the contents of an encrypted message as difficult to predict as possible.
- Put the most random data at the front of the message so that the cipher block chaining will randomize the encrypted text as much as possible.
- Do not repeat an encrypted message more often that strictly necessary.
- Do not provide “free” samples of encrypted text.

Chapter 3

Authentication Server **Interface**

3.1 Introduction

This chapter documents the API for the Authentication Server facility, as defined by the *kauth.rg Rxgen* interface file and the *kaserver.h* include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer are included.

3.2 Constants

This section covers the basic constant definitions of interest to the AuthServer application programmer. These definitions appear in the *kaserver.h* file which is automatically generated from *kauth.rg*, the RPC interface definition file. See Chapter 6 of the *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility* for an example of an RPC interface definition file.

The subsections describe constants in the following categories:

- Interface function opcodes
- Constants
- Micellaneous

3.2.1 Interface Function Opcodes

These constants, appearing in the *kauth.rg Rxgen* interface file for the AuthServer, define the opcodes for the RPC routines. Every *Rx* call to this interface contains this opcode. The thread dispatcher uses it to select the proper code at the server site to carry out the call.

The functions are described in section 3.5.

Name	*Value*	*Description*
AUTHENTICATE_OLD	1	Opcode for *KAA_Authenticate_old()*
CHANGEPASSWORD	2	Opcode for *KAA_ChangePassword()*
GETTICKET_OLD	3	Opcode for *KAT_GetTicket_old()*
SETPASSWORD	4	Opcode for *KAM_SetPassword()*
SETFIELDS	5	Opcode for *KAM_SetFields()*
CREATEUSER	6	Opcode for *KAM_CreateUser()*
DELETEUSER	7	Opcode for *KAM_DeleteUser()*
GETENTRY	8	Opcode for *KAM_GetEntry()*
LISTENTRY	9	Opcode for *KAM_ListEntry()*
GETSTATS	10	Opcode for *KAM_GetStats()*
DEBUG	11	Opcode for *KAM_Debug()*
GETPASSWORD	12	Opcode for *KAM_GetPassword()*
GETRANDOMKEY	13	Opcode for *KAM_GetRandomKey()*
AUTHENTICATE	21	Opcode for *KAA_Authenticate()*
GETTICKET	23	Opcode for *KAT_GetTicket()*

3.2.2 Constants

The following constants are required to properly use the AuthServer RPC interface, both to provide values and to interpret information returned by the calls.

Name	*Value*	*Description*
MAXKAKVNO	127	The key version number must fit in a byte.

The next six constants are flags. NOTE: zero is an illegal value.

Name	*Value*	*Description*
KAFNORMAL	0x001	Set for all user entries.

If the normal flag is off, then one of the two flags below must be set.

Name	*Value*	*Description*
KAFFREE	0x002	Set if in free list.
KAFOLDKEYS	0x010	Set if entry used to store old keys.

Otherwise one of the following may be set to define the usage of the misc field.

Name	*Value*	*Description*
KAFSPECIAL	0x100	Set if special AuthServer principal.
KAFASSOCROOT	0x200	Set if root of associate tree.
KAFASSOC	0x400	Set if entry is an associate.

The following flags define special properties for normal users and are settable using *SetFields()*.

Name	*Value*	*Description*
KAFADMIN	0x004	An administrator.
KAFNOTGS	0x008	Don't allow principal to get or use TGT.
KAFNOSEAL	0x020	Don't allow principal as server in GetTicket.
KAFNOCPW	0x040	Don't allow principal to change its own key.
KAFNEWASSOC	0x080	Allow user to create associates.

3.2.3 Miscellaneous

This section lists miscellaneous constants used in the structures described in the following sections.

To make future revisions easy to accommodate they are assigned a major and minor version number. Major version changes will require recompilation because the structures have changed size. Minor version changes will be more or less upward compatible.

Name	*Value*	*Description*
KAMAJORVERSION	5	
KAMINORVERSION	1	
NEVERDATE	037777777777	Initial value used in date fields.
KADEBUGKCINFOSIZE	25	This returns information about the state of the server for debugging problems remotely.

3.3 Structures

This section describes the major exported *Authentication Server* data structures of interest to application programmers. These structures are returned by server RPC interface routines.

3.3.1 `struct ka_CBS`

This structure is used to specify input byte sequences in routines such as *ChangePassword()* and *GetTicket()*. Since the byte sequence is not NULL terminated, the number of bytes in the sequence must be included in the structure.

Fields

long SeqLen - Number of bytes in character string.

char *SeqBody - A character string, not NULL terminated.

3.3.2 `struct ka_BBS`

This structure is used to specify where a called function, such as *ChangePassword()* and *GetTicket()*, should put a reply byte sequence and the maximum number of bytes allowed in the reply. The called function specifies the number of bytes being returned.

Fields

long MaxSeqLen - Maximum number of bytes allowed in character string.

long SeqLen - Actual number of bytes in character string.

char *SeqBody - Character string, not NULL terminated.

3.3.3 struct EncryptionKey

This structure defines an encryption key that is bit level compatible with DES and ktc_encryptionKey, but that will have to be cast to the appropriate type in calls.

Fields

char key[8] - 16 hex digits.

3.3.4 struct kaident

This structure returns name and instance strings.

Fields

`char name[MAXKANAMELEN]` - The users name, NULL terminated, at most 63 characters in name.

`char instance[MAXKANAMELEN]` - The group name, NULL terminated, at most 63 characters in group.

3.3.5 struct kaentryinfo

This structure is used to return the information stored in an entry in the authentication database. For example, the calling sequence to *GetEntry()* specifies a pointer to one of these. NOTE: `Date` is a synonym for `unsigned long`.

Fields

`long minor_version` - The minor version number.

`long flags` - Holds flags described in section 3.2.2.

`Date user_expiration` - User registration good until then.

`Date modification_time` - Time of last update.

`struct kaident modification_user` - User name and instance last modified.

`Date change_password_time` - Time user changed own password.

`long max_ticket_lifetime` - Maximum lifetime for tickets.

`long key_version` - Version number of this key.

`EncryptionKey key` - The key to use.

`unsigned long keyCheckSum` - Crypto-cksum of key.

`long reserved2` - Not used.

`long reserved3` - Not used.

`long reserved4` - Not used.

3.3.6 `struct kasstats`

These are (static) statistics kept in the database header. This structure can be examined via a call to *GetStats()*.

Fields

`long minor_version` - The minor version number.

`long allocs` - Total number of calls to AllocBlock.

`long frees` - Total number of calls to FreeBlock.

`long cpws` - Number of user change password commands.

`long reserved1` - Not used.

`long reserved2` - Not used.

`long reserved3` - Not used.

`long reserved4` - Not used.

3.3.7 struct kadstats

These are dynamic statistics kept in each AuthServer process. The current values can be examined via a call to *GetStats()*.

Fields

`long minor_version` - The minor version number.

`long host` - Server, in NW byte order.

`Date start_time` - Time statistics were last cleared.

`long hashTableUtilization` - Use of non-empty hash table entries in parts per 10,000.

`declare_stat (Authenticate)` - Count of requests and aborts for each RPC.

`declare_stat (ChangePassword)` - Count of requests and aborts for each RPC.

`declare_stat (GetTicket)` - Count of requests and aborts for each RPC.

`declare_stat (CreateUser)` - Count of requests and aborts for each RPC.

`declare_stat (SetPassword)` - Count of requests and aborts for each RPC.

`declare_stat (SetFields)` - Count of requests and aborts for each RPC.

`declare_stat (DeleteUser)` - Count of requests and aborts for each RPC.

`declare_stat (GetEntry)` - Count of requests and aborts for each RPC.

`declare_stat (ListEntry)` - Count of requests and aborts for each RPC.

`declare_stat (GetStats)` - Count of requests and aborts for each RPC.

`declare_stat (GetPassword)` - Count of requests and aborts for each RPC.

`declare_stat (GetRandomKey)` - Count of requests and aborts for each RPC.

`declare_stat (Debug)` - Count of requests and aborts for each RPC.

`declare_stat (UAuthenticate)` - Count of requests and aborts for each RPC.

`declare_stat (UGetTicket)` - Count of requests and aborts for each RPC.

`long string_checks` - Errors detected in name and instance strings.

`long reserved1` - Not used.

`long reserved2` - Not used.

`long reserved3` - Not used.

`long reserved4` - Not used.

3.3.8 struct ka_kcInfo

This structure describes an entry in the key cache in the AuthServer. This information is visible as part of the `ka_debugInfo` structure.

Fields

`Date used` - When key used last.

`long kvno` - Version number of this key.

`char primary` - Determines whether this key is due to be superseded.

`char keycksum` - The checksum for this key.

`char principal[64]` - Stores user name and group.

3.3.9 struct ka_debugInfo

This structure describes the state of the AuthServer. The information is visible via a call to *Debug()*.

Fields

`long minorVersion` - The minor version number.

`long host` - Server in NW byte order.

`Date startTime` - Time server was started.

`int noAuth` - Running with authentication off.

`Date lastTrans` - Time of last transaction.

`char lastOperation[16]` - Name of last operation.

`char lastAuth[256]` - Last principal to authenticate.

`char lastUAuth[256]` - Last principal to authenticate via UDP.

`char lastTGS[256]` - Last principal to call ticket granting service.

`char lastUTGS[256]` - Last principal to call TGS via UDP.

`char lastAdmin[256]` - Last principal to call admin service.

`char lastTGSServer[256]` - Last server for which a ticket was requested.

`char lastUTGSServer[256]` - Last server for which a ticket was requested via UDP.

`Date nextAutoCPW` - When server password expires (probably NEVERDATE).

`int updatesRemaining` - Update necessary for next AutoCPW.

`Date dbHeaderRead` - Time cheader structure was last read in.

`long dbVersion` - Minor version number.

`long dbFreePtr` - Pointer to first free entry in the database file.

`long dbEofPtr` - Pointer to first free byte in the database file.

`long dbKvnoPtr` - Pointer to first key version number in the database.

`long dbSpecialKeysVersion` - Latest key version number.

`long cheader_lock` - Lock for the key cache header.

`long keycache_lock` - Lock for the key cache.

`long kcVersion` - Minor version number.

`int kcSize` - Size of key cache.

`int kcUsed` - Number of entries in key cache.

`struct ka_kcInfo kcInfo[KADEBUGKCINFOSIZE]` - Describes the keys in the key cache.

`long reserved1` - Not used.

`long reserved2` - Not used.

`long reserved3` - Not used.

`long reserved4` - Not used.

3.4 User-Level Library Interface

The authentication functions described in the following sections are made visible via *libkauth.a*. This library in turn depends on two additional libraries: *libauth.a* and *librxkad.a*. This section lists the routines that are visible in these libraries. The programmer should use the high level routines available in *libkauth.a*.

```
libkauth.a
        KAA_Authenticate
        KAA_Authenticate_old
        KAA_ChangePassword
        KAM_CreateUser
        KAM_Debug
        KAM_DeleteUser
        KAM_GetEntry
        KAM_GetPassword
```

```
KAM_GetRandomKey
KAM_GetStats
KAM_ListEntry
KAM_SetFields
KAM_SetPassword
KAT_GetTicket
KAT_GetTicket_old
Andrew_StringToKey
StringToKey
ka_Init
ka_ParseLoginName
ka_ReadPassword
ka_StringToKey
debugCell
ka_AuthServerConn
ka_Authenticate
ka_ChangePassword
ka_ExplicitCell
ka_GetSecurity
ka_GetServers
ka_GetToken
ka_SingleServerConn
ka_GetAdminToken
ka_GetAuthToken
ka_GetServerToken
ka_CellConfig
ka_CellToRealm
ka_ExpandCell
ka_LocalCell
ka_UserAuthenticate
ka_UserAuthenticateGeneral
ka_UserReadPassword
```

libauth.a

```
afsconf_CellApply
afsconf_Check
afsconf_Close
afsconf_CloseInternal
afsconf_GetCellInfo
afsconf_GetLocalCell
afsconf_Open
afsconf_OpenInternal
afsconf_Reopen
afsconf_Touch
afsconf_AddKey
afsconf_DeleteKey
afsconf_GetKey
afsconf_GetKeys
afsconf_GetLatestKey
afsconf_IntGetKeys
afsconf_ResetKeys
afsconf_AddUser
afsconf_DeleteUser
```

```
        afsconf_GetNoAuthFlag
        afsconf_GetNthUser
        afsconf_SetNoAuthFlag
        afsconf_SuperUser
        afsconf_SetCellInfo
        afsconf_CheckAuth
        afsconf_ClientAuth
        afsconf_ClientAuthSecure
        afsconf_ServerAuth

librxkad.a

        rxkad_AllocCID
        rxkad_NewClientSecurityObject
        rxkad_client_init
        rxkad_GetServerInfo
        rxkad_NewServerSecurityObject
        rxkad_CheckPacket
        rxkad_CksumChallengeResponse
        rxkad_Close
        rxkad_DeriveXORInfo
        rxkad_DestroyConnection
        rxkad_GetStats
        rxkad_NewConnection
        rxkad_PreparePacket
        rxkad_SetLevel
        rxkad_SetupEndpoint
        ktohl
        tkt_CheckTimes
        tkt_DecodeTicket
        tkt_MakeTicket
        fc_cbc_encrypt
        fc_ecb_encrypt
        fc_keysched
        fcrypt_init
        rxkad_DecryptPacket
        rxkad_EncryptPacket
        rxkad_crypt_init
```

3.5 RPC Function Calls

This section describes the Remote Procedure Calls used between the AuthServer and clients. Each function in this section has an assigned *opcode* number, see section 3.2.1. This is the low-level numerical identifier for the function, and appears in the set of network packets constructed for the RPC call.

Authentication Server Interface

3.5.1 KAA_Authenticate — Authenticate user password

```
proc KAA_Authenticate(IN kaname name, IN kaname instance, IN Date start_time, IN Date end_time,
                      IN struct ka_CBS *request, INOUT struct ka_BBS *answer) = 21;
```

Description

Calling this routine invokes the authentication protocol described in chapter 2. It uses the `name` and `instance` to look up the user's key in the authentication database. (NOTE: The value of `instance` is always NULL.) The key allows the `request` to be decrypted and, if it is properly formed, a ticket is created. The ticket and newly invented session key are assembled into the `answer`, encrypted with the key, and returned. This ticket is referred to as the AuthTicket. The lifetime of the ticket is specified by `start_time` and `end_time`, although the actual expiration time may be earlier if the requested interval exceeds the `max_ticket_lifetime` field of the user's AuthServer entry. This request does not modify the database.

The function *KAA_Authenticate_old()* is provided for backward compatibility with old kauth.

3.5.2 KAA_ChangePassword — Change user password

```
proc KAA_ChangePassword(IN kaname name, IN kaname instance, IN struct ka_CBS *arequest,
                        INOUT struct ka_BBS *oanswer) = 2;
```

Description

This call invokes the protocol for changing passwords described above. The encryption key used for both the *request* and *answer* sequences is the old one. The new key takes effect as soon as the lock associated with this operation is released. This request modifies the authentication database, but instead of updating the modification data a separate field called `change_password_time` is set.

3.5.3 KAT_GetTicket — Ask kaserver for a ticket for some other service

```
proc KAT_GetTicket(IN long kvno, IN kaname auth_domain, IN struct ka_CBS *aticket,
                   IN kaname name, IN kaname instance, IN struct ka_CBS *atimes,
                   INOUT struct ka_BBS *oanswer) = 23;
```

Description

This call requires an RPC connection encrypted with the AuthTicket. As long as that ticket is valid, the `name` and `instance` of a server are used to create a ticket and associated session key for that server. (NOTE: The value of `instance` is always NULL.) This operation does not modify the database.

The routine *ka_GetToken()* performs the same function and may be easier for the application writer to use.

The function *KAT_GetTicket_old()* is provided for backward compatibility with old kauth.

3.5.4 KAM_SetPassword — Initialize a password

```
proc KAM_SetPassword(IN kaname name, IN kaname instance, IN long kvno,
                     IN EncryptionKey password) = 4;
```

Description

The key and key version number of the user are set to the provided values. This call requires an RPC connection encrypted with an AdminTicket.

3.5.5 KAM_SetFields — Reset values in user database entry

```
proc KAM_SetFields(IN kaname name, IN kaname instance, IN long flags,
                   IN Date user_expiration, IN long max_ticket_lifetime, IN long maxAssociates,
                   IN long spare1, IN long spare2) = 5;
```

Description

This function alters the miscellaneous parameters associated with a user. The `flags` field can be set to one of three values.

- Normal - This is the default state: a regular user.
- Admin - This user is privileged and can modify the authentication database.
- Inactive - This makes the entry a placeholder. The user is not deleted but authentication attempts will fail.

The `user_expiration` is the time after which attempts to authenticate as this user will fail. The `max_ticket_lifetime` can be set to limit the lifetime of an authentication ticket created for a user. This call requires an RPC connection encrypted with an AdminTicket.

3.5.6 KAM_CreateUser — Enter a user in the database

`proc` **KAM_CreateUser**`(IN kaname name, IN kaname instance, IN EncryptionKey password) = 6;`

Description

This function adds a user to the authentication database. The key version number will be zero. The user's flags and maximum ticket lifetime will be set to default values. The registration will not have an expiration time. The modification data is set. This call requires an RPC connection encrypted with an AdminTicket.

3.5.7 KAM_DeleteUser — Delete a user from the database

`proc` **KAM_DeleteUser** `(IN kaname name, IN kaname instance) = 7;`

Description

This function removes a user from the authentication database. It requires an RPC connection encrypted with an AdminTicket.

3.5.8 KAM_GetEntry — Return a user entry in the database

```
proc KAM_GetEntry(IN kaname name, IN kaname instance, IN long major_version,
                  OUT struct kaentryinfo *entry) = 8;
```

Description

This function returns information about an entry in the authentication database. If the major version number does not match that in use by the server, the call returns an error code. This request does not modify the database. This call requires an RPC connection encrypted with an AdminTicket.

3.5.9 KAM_ListEntry — List entries in sequence

```
proc KAM_ListEntry(IN long previous_index,
                   OUT long *index, OUT long *count, OUT kaident *name) = 9;
```

Description

This function provides a way to step through all the entries in the database. The first call should be made with `previous_index` set to zero. The function returns `count`, which is an estimate of the number of entries remaining to be returned, and `index`, which should be passed in as `previous_index` on the next call. Each call that returns a non-zero `index` also returns a structure `kaident`, which gives the name and instance of an entry. A negative `count` or a non-zero return code indicates that an error occurred. A zero `index` means there were no more entries. A zero `count` means the last entry has been returned. This call does not modify the database and requires an RPC connection encrypted with an AdminTicket.

3.5.10 KAM_GetStats — Returns kasstats and kadstats

```
proc KAM_GetStats(IN long major_version, OUT long *admin_accounts,
                  OUT struct kasstats *statics, OUT struct kadstats *dynamics) = 10;
```

Description

This function returns statistics about the AuthServer and its database. If the `major_version` does not match that used by the server, the call returns an error code. The database is not modified. This call requires an RPC connection encrypted with an AdminTicket.

3.5.11 KAM_Debug — Return ka_debugInfo

```
proc KAM_Debug(IN long major_version, IN int checkDB, OUT struct ka_debugInfo *info) = 11;
```

Description

This function returns information about the authentication database, the key cache, and the state of the AuthServer.

3.5.12 KAM_GetPassword — Return a users password

```
proc KAM_GetPassword (IN kaname name, OUT EncryptionKey *password) = 12;
```

Description

This function returns a password from an entry in the authentication database. This could be either a user password (by specifying a user name) or a server password (by specifying a service name, e.g., "afs").

3.5.13 KAM_GetRandomKey — Return a random legal DES key

```
proc KAM_GetRandomKey (OUT EncryptionKey *password) = 13;
```

Description

This function returns a random DES key and is preferred over a calling routine just inventing a key. It returns a properly formatted 8 byte DES key, sets the parity properly in each byte of the key, and avoids the 5 "bad" keys.

3.6 User-Level Library Functions

This section contains specifications for the remaining routines that are visible in *libkauth.a*. These routines are the higher level interface that is most frequently used by management applications.

3.6.1 ka_UserAuthenticateGeneral — Authenticate a user

```
long ka_UserAuthenticateGeneral(long flags,
                                char *name,
                                char *instance,
                                char *realm,
                                char *password,
                                Date lifetime,
                                long spare1,
                                long spare2,
                                char **reasonP)
```

Description

This function returns 0 if successful. `name`, `instance` and `realm` can be gotten from a principal by calling *ka_ParseLoginName()*. `reason` will contain a textual description of the failure in case of error `spare2` must be 0.

This routine replaces *ka_UserAuthenticate()*, which is retained in the library for backward compatibility.

3.6.2 ka_UserReadPassword — Read a user password

```
long ka_UserReadPassword (char *prompt,
                          char *password,
                          int plen,
                          char **reasonP)
```

Description

This routine does a non-echo read of a password entered from a prompt on a standard tty. Error messages are returned in `reasonP`.

3.7 Cell Related Functions

The following routines are used by management applications to manage cell information.

3.7.1 ka_CellConfig — Access the cell configuration directory

```
int ka_CellConfig(char *dir)
```

Description

This function uses `dir` to find the specified *CellServDB* file, otherwise it uses */usr/vice/etc/CellServDB* as the default.

3.7.2 ka_LocalCell — Return the local cell name

```
char *ka_LocalCell()
```

Description

This function returns 0 as an error, otherwise it returns the local cell name.

3.7.3 ka_ExpandCell — Expands short cell name

```
int ka_ExpandCell(char *cell,
                  char *fullCell,
                  int *alocal)
```

Description

`cell` and `fullCell` must be preallocated or else NULL. `cell` is a prefix of a cell name. If `cell` is 0, the name of the local cell will be used. `fullCell` and `alocal` are OUT parameters. `fullCell` is the expanded cell name which matches the prefix. `alocal` indicates whether the returned cell name is the name of the local cell. If a cell name is not returned then the function returns `KANOCELL`, `KANOCELLS` or 0.

3.7.4 ka_CellToRealm — Convert cell name to upper case

```
int ka_CellToRealm(char *cell,
                   char *realm,
                   int *local)
```

Description

This function converts a cell name into a Kerberos realm name. It actually calls *ka_ExpandCell()* and converts the returned cell name into upper case.

Like *kaExpandCell()*, if `cell` is 0 it uses the local cell name and sets `local` to 1. Otherwise it uses the prefix found in `cell` to determine the full cell name. If the cell name is not local then `local` is set to 0. The expanded cell name is returned in `realm`. If a cell name is not returned then the function returns `KANOCELL`, `KANOCELLS` or 0.

3.8 Miscellaneous Client-Side Functions

This section contains descriptions of a variety of client functions.

The first three functions are called from management applications to manage passwords and DES keys. They accept a password string as input and convert it via a one-way encryption algorithm to a DES encryption key. In all three `str`, `cell` and `key` must be pre-allocated and null-terminated. *strlen(cell)* must be less than `MAXKTCREALMLEN`.

3.8.1 ka_Andrew_StringToKey — Convert password to DES key

```
static void ka_AndrewStringToKey(char *str,
                                 char *cell,
                                 struct ktc_encryptionKey *key)
```

Description

This function is compatible with the original Andrew authentication service password database.

3.8.2 StringToKey — Convert password to DES key

```
static void StringToKey(char *str,
                        char *cell,
                        struct ktc_encryptionKey *key)
```

3.8.3 ka_StringToKey — Convert password to DES key

```
int ka_StringToKey(char *str
                   char *cell,
                   struct ktc_encryptionKey *key)
```

3.8.4 ka_ReadPassword — Read password, convert to DES key

```
long ka_ReadPassword(char *prompt,
                     int verify,
                     char *cell,
                     struct ktc_encryptionKey *key)
```

Description

This function prints out a prompt and reads a string from the terminal, turning off echoing. If `verify` is requested it requests that the string be entered again and the two strings are compared. The string is then converted to a DES encryption key. A non-zero return indicates an error while reading in the password string.

3.8.5 ka_ParseLoginName — Parse user-entered login name

```
long ka_ParseLoginName(char *login,
                       char name[MAXKTCNAMELEN],
                       char inst[MAXKTCNAMELEN],
                       char cell[MAXKTCREALMLEN])
```

Description

This routine parses a string that might be entered by a user from the terminal. It defines a syntax that allows a user to specify his or her identity in terms of his or her name, instance and cell with a single string. These three output strings must be allocated by the caller to their maximum length. The syntax is very simple: the first dot ('.') separates the name from the instance and the first atsign ('@') begins the cell name. A backslash ('\') can be used to quote these special characters. A backslash followed by an octal digit (zero through seven) introduces a three digit octal number which is interpreted as the ascii value of a single character. A non-zero return indicates an argument error.

Error Codes

`KABADARGUMENT` No output parameters were specified.

`KABADNAME` A component of the user name was too long or a cell was specified but the cell parameter was null.

3.8.6 ka_Init — Initialize a client to make calls to the AuthServer

```
long ka_Init(int flags)
```

Client side applications call this function to initialize error tables and connect to the correct *CellServDB* file. The argument is not used but reserved for future use.

3.8.7 ka_ExplicitCell — Copy list of servers to debugging cell

```
void ka_ExplicitCell(char *cell,
                     long serverList[])
```

This function copies the specified list of servers in `serverList` into a specially known cell named "explicit". The cell can then be used to debug experimental servers.

3.8.8 ka_GetServers — Return a list of AuthServers

```
int ka_GetServers(char *cell,
                  struct afsconf_cell *cellinfo)
```

This function returns a list of Authentication Servers in the cell named in `cell`.

3.8.9 ka_GetSecurity — Return a security object

```
long ka_GetSecurity(int service,
                    struct ktc_token *token,
                    struct rx_securityClass **scP,
                    int *siP)
```

This function returns a security object appropriate for the `service` passed in as an argument.

Error Codes

`KABADARGUMENT` `service` was not recognized.

`KARXFAIL` Failed to get security object.

3.8.10 ka_SingleServerConn — Establish connection to one server

```
long ka_SingleServerConn(char *cell,
                         char *server,
                         int service,
                         struct ktc_token *token,
                         struct ubik_client **conn)
```

This function establishes a connection to only one Authentication Server. Inputs to the function are

- `cell` - the cell name
- `server` - the server for which the connection is being requested
- `service` - the service required
- `token` - a token may or may not be needed depending on the service.

The connection is returned in `conn`.

3.8.11 ka_AuthServerConn — Establish connection to all AuthServers

```
long ka_AuthServerConn(char *cell,
                       int service,
                       struct ktc_token *token,
                       struct ubik_client **conn)
```

Description

This function gets connections to all the Authentication Servers in a cell. Inputs to the function are

- `cell` - the cell name
- `service` - the service required
- `token` - a token may or may not be needed depending on the service.

The connection is returned in `conn`.

3.8.12 ka_Authenticate — Return a ticket for the ticket-granting service

```
long ka_Authenticate(char *name,
                     char *instance,
                     struct ubik_client *conn,
                     int service,
                     struct ktc_encryptionKey *key,
                     Date start,end,
                     struct ktc_token *token)
```

This function is the interface to the AuthServer RPC routine *KAA_Authenticate()*. It formats the request packet, calls the encryption routine on the answer, calls *KAA_Authenticate()* and decrypts the response. The response is checked for correctness and its contents are copied into the token. A non-zero return indicates either bad key parity, a timeout, a bad packet or a server error.

3.8.13 ka_ChangePassword — Change user password in the database

```
long ka_ChangePassword(char *name,
                       char *instance,
                       struct ubik_client *conn,
                       struct ktc_encryptionKey *oldkey,
                       struct ktc_encryptionKey *newkey)
```

3.8.14 ka_GetToken — Return a ticket for the named service

```
long ka_GetToken(char *name,
                 char *instance,
                 struct ubik_client *conn,
                 Date start, end,
                 struct ktc_token *auth_token,
                 char *auth_domain,
                 struct ktc_token *token)
```

This is a low-level function used by *ka_GetServerToken()*. It assumes that a connection to the AuthServer has already been established. The desired ticket lifetime is specified by `start` and `end`.

3.8.15 ka_GetAdminToken — Return an admin token

```
long ka_GetAdminToken(char *name,
                      char *instance,
                      char *cell,
                      struct ktc_encryptionKey *key,
                      long lifetime,
                      struct ktc_token *token,
                      int new)
```

Description

`key` contains the key constructed from the user's password and `lifetime` indicates how long the ticket will be valid (in seconds). `token` is used to pass in a token and to return one. If `token` is NULL, a token is not returned. If `new` is set to 1, then the function should get a new token if necessary.

3.8.16 ka_GetAuthToken — Return an authentication token

```
long ka_GetAuthToken(char *name,
                     char *instance,
                     char *cell,
                     struct ktc_encryptionKey *key;
                     long lifetime)
```

Description

This function returns a ticket to administer for a particular principle specified by `name` and `instance`. `key` contains the key constructed from the user's password and `lifetime` indicates how long the key will be valid (in seconds).

3.8.17 ka_GetServerToken — Return a token for a server

```
long ka_GetServerToken(char *name,
                       char *instance;
                       char *cell;
                       Date lifetime;
                       struct ktc_token *token;
```

This function sets up the necessary connections etc. and then calls *ka_GetToken()*.

3.9 Command Line Interface

Another interface exported by the *Authentication Server* is the set of command line switches it accepts. Using these switches, many server parameters and actions can be set. Under normal conditions, the *Authentication Server* process is started up by the *BOS Server* on that machine, as described in *AFS-3 Programmer's Reference: BOS Server Interface*. So, in order to utilize any combination of these command-line options, the system administrator must define the *Authentication Server* bnode in such a way that these parameters are properly included.

A description of the set of currently-supported command line switches can be found in the *AFS 3.0 Command Reference Manual*.

Bibliography

[1] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[2] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[3] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[4] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*, RFC 1094, March 1989.

[5] Sun Microsystems, Inc. *Design and Implementation of the Sun Network File System*, USENIX Summer Conference Proceedings, June 1985.

[6] S.P. Miller, B.C. Neuman, J.I. Schiller, J.H. Saltzer. *Kerberos Authentication and Authorization System*, Project Athena Technical Plan, Section E.2.1, M.I.T., December 1987.

[7] R. Needham, M. Schroeder. *Using Encryption for Authentication in Large Networks of Computers*, CACM, 21(1978)993-9.

[8] S. Matyas, C. Meyer. *Generation, Distribution and Installation of Cryptographic Keys*, IBM Sys Jr, 17(1978)126-37.

BOS Server Specification

Chapter 4

Overview

4.1 Introduction

One of the important duties of an AFS system administrator is to insure that processes on file server machines are properly installed and kept running. The *BOS Server* was written as a tool for assisting administrators in these tasks. An instance of the *BOS Server* runs on each AFS server machine, and has the following specific areas of responsibility:

- **Definition of the set of processes that are to be run** on the machine on which a given *BOS Server* executes. This definition may be changed dynamically by system administrators. Programs may be marked as continuously or periodically runnable.
- **Automatic startup and restart of these specified processes** upon server bootup and program failure. The *BOS Server* also responds to administrator requests for stopping and starting one or more of these processes. In addition, the *BOS Server* is capable of restarting itself on command.
- **Collection of information** regarding the current status, command line parameters, execution history, and log files generated by the set of server programs.
- **Management of the security information** resident on the machine on which the *BOS Server* executes. Such information includes the list of administratively privileged people associated with the machine and the set of AFS *File Server* encryption keys used in the course of file service.

- **Management of the cell configuration information** for the server machine in question. This includes the name of the cell in which the server resides, along with the list and locations of the servers within the cell providing AFS database services (e.g., volume location, authentication, protection).
- **Installation of server binaries** on the given machine. The *BOS Server* allows several "generations" of server software to be kept on its machine. Installation of new software for one or more server agents is handled by the *BOS Server*, as is "rolling back" to a previous version should it prove more stable than the currently-installed image.
- **Execution of commands on the server machine**. An administrator may execute arbitrary UNIX commands on a machine running the *BOS Server*.

Unlike many other AFS server processes, the *BOS Server* does *not* maintain a cell-wide, replicated database. It does, however, maintain several databases used exclusively on every machine on which it runs.

4.2 Scope

This paper describes the design and structure of the AFS-3 *BOS Server*. The scope of this work is to provide readers with a sufficiently detailed description of the *BOS Server* so that they may construct client applications that call the server's RPC interface routines.

4.3 Document Layout

The second chapter discusses various aspects of the *BOS Server*'s architecture. First, one of the basic concepts is examined, namely the **bnode**. Providing the complete description of a program or set of programs to be run on the given server machine, a bnode is the generic definitional unit for the *BOS Server*'s duties. After bnodes have been explained, the set of standard directories on which the *BOS Server* depends is considered. Also, the set of well-known files within these directories is explored. Their uses and internal formats are presented. After these sections, a discussion of *BOS Server* restart times follows. The *BOS Server* has special support for two commonly-used restart occasions, as described by this section. Finally, the organization and behavior of the *bosserver* program itself is presented.

The third and final chapter provides a detailed examination of the programmer-visible *BOS Server* constants and structures, along with a full specification of the API for the RPC-based *BOS Server* functionality.

4.4 Related Documents

This document is a member of a documentation suite providing programmer-level specifications for the operation of the various AFS servers and agents, and the interfaces they export, as well as the underlying RPC system they use to communicate. The full suite of related AFS specification documents is listed below:

- *AFS-3 Programmer's Reference: Architectural Overview*: This paper provides an architectual overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and examining their interactions.
- *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*: This document describes the *File Server* and *Cache Manager* agents, which provide the backbone file managment services for AFS. The collection of *File Server*s for a cell supply centralized file storage for that site, and allow clients running the *Cache Manager* component to acces those files in a high-performance, secure fashion.
- *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.
- *AFS-3 Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for mapping printable user names to and from their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users, which are suitable for placement on ACLs.
- *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*: This document specifies the design and operation of the remote procedure call and lightweight process packages used by AFS.

In addition to these papers, the AFS 3.1 product is delivered with its own user, administrator, installation, and command reference documents.

Chapter 5

BOS Server **Architecture**

This chapter considers some of the architectual features of the AFS-3 *BOS Server*. First, the basic organizational and functional entity employed by the *BOS Server*, the **bnode**, is discussed. Next, the set of files with which the server interacts is examined. The notion of **restart times** is then explained in detail. Finally, the organization and components of the *bosserver* program itself, which implements the *BOS Server*, are presented.

5.1 Bnodes

5.1.1 Overview

The information required to manage each AFS-related program running on a *File Server* machine is encapsulated in a **bnode** object. These bnodes serve as the basic building blocks for *BOS Server* services. Bnodes have two forms of existence:

- **On-disk**: The *BosConfig* file (see Section 5.3.4 below) defines the set of bnodes for which the *BOS Server* running on that machine will be responsible, along with specifying other information such as values for the two restart times. This file provides permanent storage (i.e., between bootups) for the desired list of programs for that server platform.
- **In-memory**: The contents of the *BosConfig* file are parsed and internalized by the *BOS Server* when it starts execution. The basic data for a particular server program is placed into a `struct bnode` structure.

The initial contents of the *BosConfig* file are typically set up during system installation. The *BOS Server* can be directed, via its RPC interface, to alter existing bnode entries in the *BosConfig* file, add new ones, and delete old ones. Typically, this file is never edited directly.

5.1.2 Bnode Classes

The descriptions of the members of the AFS server suite fall into three broad classes of programs:

- **Simple programs**: This server class is populated by programs that simply need to be kept running, and do not depend on other programs for correctness or effectiveness. Examples of AFS servers falling into this category are the *Volume Location Server*, *Authentication Server*, and *Protection Server*. Since its members exhibit such straightforward behavior, this class of programs is referred to as the **simple** class.

- **Interrelated programs**: The *File Server* program depends on two other programs, and requires that they be executed at the appropriate times and in the appropriate sequence, for correct operation. The first of these programs is the *Volume Server*, which must be run concurrently with the *File Server*. The second is the *salvager*, which repairs the AFS volume metadata on the server partitions should the metadata become damaged. The *salvager* **must not** be run at the same time as the *File Server*. In honor of the *File Server* trio that inspired it, the class of programs consisting of groups of interrelated processes is named the **fs** class.

- **Periodic programs**: Some AFS servers, such as the *BackupServer*, only need to run every so often, but on a regular and well-defined basis. The name for this class is taken from the UNIX tool that is typically used to define such regular executions, namely the **cron** class.

The recognition and definition of these three server classes is exploited by the *BOS Server*. Since all of the programs in a given class share certain common characteristics, they may all utilize the same basic data structures to record and manage their special requirements. Thus, it is not necessary to reimplement all the operations required to satisfy the capabilities promised by the *BOS Server* RPC interface for each and every program the *BOS Server* manages. Implementing one set of operations for each server class is sufficient to handle any and all server binaries to be run on the platform.

5.1.3 Per-Class Bnode Operations

As mentioned above, only one set of basic routines must be implemented for each AFS server class. Much like Sun's *VFS/vnode* interface [8], providing a common set of routines for interacting with a given file system, regardless of its underlying implementation and semantics, the *BOS Server* defines a common vector of operations for a class of programs to be run under the *BOS Server*'s tutelage. In fact, it was this standardized file system interface that inspired the "bnode" name.

The *BOS Server* manipulates the process or processes that are described by each bnode by invoking the proper functions in the appropriate order from the operation vector for that server class. Thus, the *BOS Server* itself operates in a class-independent fashion. This allows each class to take care of the special circumstances associated with it, yet to have the *BOS Server* itself be totally unaware of what special actions (if any) are needed for the class. This abstraction also allows more server classes to be implemented without any significant change to the *BOS Server* code itself.

There are ten entries in this standardized class function array. The purpose and usage of each individual class function is described in detail in Section 6.3.5. Much like the VFS system, a collection of macros is also provided in order to simplify the invocation of these functions. These macros are described in Section 6.5. The ten function slots are named here for convenience:

- *create()*
- *timeout()*
- *getstat()*
- *setstat()*
- *delete()*
- *procexit()*
- *getstring()*
- *getparm()*
- *restartp()*
- *hascore()*

5.2 *BOS Server* Directories

The *BOS Server* expects the existence of the following directories on the local disk of the platform on which it runs. These directories define where the system binaries, log files, *ubik* databases, and other files lie.

- */usr/afs/bin*: This directory houses the full set of AFS server binaries. Such executables as *bosserver*, *fileserver*, *vlserver*, *volserver*, *kaserver*, and *ptserver* reside here.
- */usr/afs/db*: This directory serves as the well-known location on the server's local disk for the *ubik* database replicas for this machine. Specifically, the *Authentication Server*, *Protection Server*, and the *Volume Location Server* maintain their local database images here.
- */usr/afs/etc*: This directory hosts the files containing the security, cell, and authorized system administrator list for the given machine. Specifically, the *CellServDB*, *KeyFile*, *License*, *ThisCell*, and *UserList* files are stored here.
- */usr/afs/local*: This directory houses the *BosConfig* file, which supplies the *BOS Server* with the permanent set of bnodes to support. Also contained in this directory are files used exclusively by the *salvager*.
- */usr/afs/logs*: All of the AFS server programs that maintain log files deposit them in this directory.

5.3 *BOS Server* Files

Several files, some mentioned above, are maintained on the server's local disk and manipulated by the *BOS Server*. This section examines many of these files, and describes their formats.

5.3.1 */usr/afs/etc/UserList*

This file contains the names of individuals who are allowed to issue "restricted" *BOS Server* commands (e.g., creating & deleting bnodes, setting cell information, etc.) on the given hardware platform. The format is straightforward, with one administrator name per line. If a cell grants `joe` and `schmoe` these rights on a machine, that particular *UserList* will have the following two lines:

```
joe
schmoe
```

5.3.2 */usr/afs/etc/CellServDB*

This file identifies the name of the cell to which the given server machine belongs, along with the set of machines on which its *ubik* database servers are running. Unlike the *CellServDB* found in */usr/vice/etc* on AFS client machines, this file contains *only* the entry for the home cell. It shares the formatting rules with the */usr/vice/etc* version of *CellServDB*. The contents of the *CellServDB* file used by the *BOS Server* on host `grand.central.org` are:

```
>grand.central.org        #DARPA clearinghouse cell
192.54.226.100            #grand.central.org
192.54.226.101            #penn.central.org
```

5.3.3 */usr/afs/etc/ThisCell*

The *BOS Server* obtains its notion of cell membership from the *ThisCell* file in the specified directory. As with the version of *ThisCell* found in */usr/vice/etc* on AFS client machines, this file simply contains the character string identifying the home cell name. For any server machine in the `grand.central.org` cell, this file contains the following:

```
grand.central.org
```

5.3.4 */usr/afs/local/BosConfig*

The *BosConfig* file is the on-disk representation of the collection of bnodes this particular *BOS Server* manages. The *BOS Server* reads and writes to this file in the normal course of its affairs. The *BOS Server* itself, in fact, should be the only agent that modifies this file. Any changes to *BosConfig* should be carried out by issuing the proper RPCs to the *BOS Server* running on the desired machine.

The following is the text of the *BosConfig* file on `grand.central.org`. A discussion of the contents follows immediately afterwards.

```
restarttime 11 0 4 0 0
checkbintime 3 0 5 0 0
bnode simple kaserver 1
parm /usr/afs/bin/kaserver
end
bnode simple ptserver 1
parm /usr/afs/bin/ptserver
end
bnode simple vlserver 1
parm /usr/afs/bin/vlserver
end
bnode fs fs 1
parm /usr/afs/bin/fileserver
parm /usr/afs/bin/volserver
parm /usr/afs/bin/salvager
end
bnode simple runntp 1
parm /usr/afs/bin/runntp -localclock transarc.com
end
bnode simple upserver 1
parm /usr/afs/bin/upserver
end
bnode simple budb_server 1
parm /usr/afs/bin/budb_server
end
bnode cron backup 1
parm /usr/afs/backup/clones/lib/backup.csh daily
parm 05:00
end
```

The first two lines of this file set the system and new-binary restart times (see Section 5.4, below). They are optional, with the default system restart time being every Sunday at 4:00am and the new-binary restart time being 5:00am daily. Following the reserved words `restarttime` and `checkbintime` are five integers, providing the mask, day, hour, minute, and second values (in decimal) for the restart time, as diagrammed below:

```
restarttime  <mask> <day> <hour> <minute> <second>
checkbintime <mask> <day> <hour> <minute> <second>
```

The range of acceptable values for these fields is presented in Section 6.3.1. In this example, the `restart` line specifies a (decimal) mask value of 11, selecting the `KTIME_HOUR`, `KTIME_MIN`, and `KTIME_DAY` bits. This indicates that the hour, minute, and day values are the ones to be used when matching times. Thus, this line requests that system restarts occur on day 0 (Sunday), hour 4 (4:00am), and minute 0 within that hour.

The sets of lines that follow define the individual bnodes for the particular machine. The first line of the bnode definition set must begin with the reserved word `bnode`, followed by the type name, the instance name, and the initial bnode goal:

```
bnode <type_name> <instance_name> <goal_val>
```

The <`type_name`> and <`instance_name`> fields are both character strings, and the <`goal_val`> field is an integer. Acceptable values for the <`type_name`> are `simple`, `fs`, and `cron`. Acceptable values for <`goal_val`> are defined in Section 6.2.3, and are normally restricted to the integer values representing `BSTAT_NORMAL` and `BSTAT_SHUTDOWN`. Thus, in the bnode line defining the *Authentication Server*, it is declared to be of type `simple`, have instance name `kaserver`, and have `1` (`BSTAT_NORMAL`) as a goal (e.g., it should be brought up and kept running).

Following the `bnode` line in the *BosConfig* file may be one or more `parm` lines. These entries represent the command line parameters that will be used to invoke the proper related program or programs. The entire text of the line after the `parm` reserved word up to the terminating newline is stored as the command line string.

```
parm <command_line_text>
```

After the `parm` lines, if any, the reserved word `end` must appear alone on a line, marking the end of an individual bnode definition.

5.3.5 */usr/afs/local/NoAuth*

The appearance of this file is used to mark whether the *BOS Server* is to insist on properly authenticated connections for its restricted operations or whether it will allow any caller to exercise these commands. Not only is the *BOS Server* affected by the presence of this file, but so are all of the bnodes objects the *BOS Server* starts up. If */usr/afs/local/NoAuth* is present, the *BOS Server* will start all of its bnodes with the `-noauth` flag.

Completely unauthenticated AFS operation will result if this file is present when the *BOS Server* starts execution. The file itself is typically empty. If any data is put into the *NoAuth* file, it will be ignored by the system.

5.3.6 */usr/afs/etc/KeyFile*

This file stores the set of AFS encryption keys used for file service operations. The file follows the format defined by `struct afsconf_key` and `struct afsconf_keys` in include file *afs/keys.h*. For the reader's convenience, these structures are detailed below:

```
#define AFSCONF_MAXKEYS 8

struct afsconf_key {
    long kvno;
    char key[8];
};

struct afsconf_keys {
    long nkeys;
    struct afsconf_key key[AFSCONF_MAXKEYS];
};
```

The first longword of the file reveals the number of keys that may be found there, with a maximum of `AFSCONF_MAXKEYS` (8). The keys themselves follow, each preceded by its key version number.

All information in this file is stored in network byte order. Each *BOS Server* converts the data to the appropriate host byte order befor storing and manipulating it.

5.4 Restart Times

It is possible to manually start or restart any server defined within the set of *BOS Server* bnodes from any AFS client machine, simply by making the appropriate call to the RPC interface while authenticated as a valid administrator (i.e., a principal listed in the *UserList* file on the given machine). However, two restart situations merit the implementation of special functionality within the *BOS Server*. There are two common occasions, occuring on a regular basis, where the entire system or individual server programs should be brought down and restarted:

- **Complete system restart**: To guard against the reliability and performance problems caused by any core leaks in long-running programs, the entire AFS system should be occasionally shut down and restarted periodically. This action "clears out" the memory system, and may result in smaller memory images for these servers, as internal data structures are reinitialized back to their starting sizes. It also allows AFS partitions to be regularly examined, and salvaged if necessary.

  Another reason for performing a complete system restart is to commence execution of a new release of the *BOS Server* itself. The new-binary restarts described below do not restart the *BOS Server* if a new version of its software has been installed on the machine.

- **New-binary restarts**: New server software may be installed at any time with the assistance of the *BOS Server*. However, it is often not the case that such software installations occur as a result of the discovery of an error in the program or programs requiring immediate restart. On these occasions, restarting the given processes in prime time so that the new binaries may begin execution is counter-productive, causing system downtime and interfering with user productivity. The system administrator may wish to set an off-peak time when the server binaries are automatically compared to the running program images, and restarts take place should the on-disk binary be more recent than the currently running program. These restarts would thus minimize the resulting service disruption.

Automatically performing these restart functions could be accomplished by creating `cron`-type bnodes that were defined to execute at the desired times. However, rather than force the system administrator to create and supervise such bnodes, the *BOS Server* supports

the notion of an internal LWP thread with the same effect (see Section 5.5.2). As part of the *BosConfig* file defined above, the administrator may simply specify the values for the complete system restart and new-binary restart times, and a dedicated *BOS Server* thread will manage the restarts.

Unless otherwise instructed, the *BOS Server* selects default times for the two above restart times. A complete system restart is carried out every Sunday at 4:00am by default, and a new-binary restart is executed for each updated binary at 5:00am every day.

5.5 The *bosserver* Process

5.5.1 Introduction

The user-space *bosserver* process is in charge of managing the AFS server processes and software images, the local security and cell database files, and allowing administrators to execute arbitrary programs on the server machine on which it runs. It also implements the RPC interface defined in the *bosint.xg Rxgen* definition file.

5.5.2 Threading

As with all the other AFS server agents, the *BOS Server* is a multithreaded program. It is configured so that a minimum of two lightweight threads are guaranteed to be allocated to handle incoming RPC calls to the *BOS Server*, and a maximum of four threads are commissioned for this task.

In addition to these threads assigned to RPC duties, there is one other thread employed by the *BOS Server*, the *BozoDaemon()*. This thread is responsible for keeping track of the two major restart events, namely the system restart and the new binary restart (see Section 5.4). Every 60 seconds, this thread is awakened, at which time it checks to see if either deadline has occurred. If the complete system restart is then due, it invokes internal *BOS Server* routines to shut down the entire suite of AFS agents on that machine and then reexecute the *BOS Server* binary, which results in the restart of all of the server processes. If the new-binary time has arrived, the *BOS Server* shuts down the bnodes for which binaries newer than those running are available, and then invokes the new software.

In general, the following procedure is used when stopping and restarting processes. First, the *restart()* operation defined for each bnode's class is invoked via the *BOP_RESTART()* macro. This allows each server to take any special steps required before cycling its service. After that function completes, the standard mechanisms are used to shut down each bnode's process, wait until it has truly stopped its execution, and then start it back up again.

5.5.3 Initialization Algorithm

This section describes the procedure followed by the *BOS Server* from the time when it is invoked to the time it has properly initialized the server machine upon which it is executing.

The first check performed by the *BOS Server* is whether or not it is running as `root`. It needs to manipulate local UNIX files and directories in which only root has been given access, so it immediately exits with an error message if this is not the case. The *BOS Server*'s UNIX working directory is then set to be */usr/afs/logs* in order to more easily service incoming RPC requests to fetch the contents of the various server log files at this location. Also, changing the working directory in this fashion results in any core images dumped by the *BOS Server*'s wards will be left in */usr/afs/logs*.

The command line is then inspected, and the *BOS Server* determines whether it will insist on authenticated RPC connections for secure administrative operations by setting up the */usr/afs/local/NoAuth* file appropriately (see Section 5.3.5). It initializes the underlying bnode package and installs the three known bnode types (`simple`, `fs`, and `cron`).

After the bnode package is thus set up, the *BOS Server* ensures that the set of local directories on which it will depend are present; refer to Section 5.2 for more details on this matter. The license file in */usr/afs/etc/License* is then read to determine the number of AFS server machines the site is allowed to operate, and whether the cell is allowed to run the NFS/AFS Translator software. This file is typically obtained in the initial system installation, taken from the installation tape. The *BOS Server* will exit unless this file exists and is properly formatted.

In order to record its actions, any existing */usr/afs/logs/BosLog* file is moved to *BosLog.old*, and a new version is opened in append mode. The *BOS Server* immediately writes a log entry concerning the state of the above set of important directories.

At this point, the *BOS Server* reads the *BosConfig* file, which lists the set of bnodes for which it will be responsible. It starts up the processes associated with the given bnodes. Once accomplished, it invokes its internal system restart LWP thread (covered in Section 5.5.2 above).

Rx initialization begins at this point, setting up the RPC infrastructure to receive its packets on the `AFSCONF_NANNYPORT`, UDP port 7007. The local cell database is then read and internalized, followed by acquisition of the AFS encryption keys.

After all of these steps have been carried out, the *BOS Server* has gleaned all of the necessary information from its environemnt and has also started up its wards. The final initialization action required is to start all of its listener LWP threads, which are devoted to executing incoming requests for the *BOS Server* RPC interface.

5.5.4 Command Line Switches

The *BOS Server* recognizes exactly one command line argument: `-noauth`. By default, the *BOS Server* attempts to use authenticated RPC connections (unless the */usr/afs/local/NoAuth* file is present; see Section 5.3.5). The appearance of the `-noauth` command line flag signals that this server incarnation is to use unauthenticated connections for even those operations that are normally restricted to system administrators. This switch is essential during the initial AFS system installation, where the procedures followed to bootstrap AFS onto a new machine require the *BOS Server* to run before some system databases have been created.

Chapter 6

BOS Server **Interface**

6.1 Introduction

This chapter documents the API for the *BOS Server* facility, as defined by the *bosint.xg Rxgen* interface file and the *bnode.h* include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer appear in this chapter.

6.2 Constants

This section covers the basic constant definitions of interest to the *BOS Server* application programmer. These definitions appear in the *bosint.h* file, automatically generated from the *bosint.xg Rxgen* interface file. Another file is exported to the programmer, namely *bnode.h*.

Each subsection is devoted to describing constants falling into each of the following categories:

- Status bits
- Bnode activity bits
- Bnode states
- Pruning server binaries

- Flag bits for `struct bnode_proc`

One constant of general utility is `BOZO_BSSIZE`, which defines the length in characters of *BOS Server* character string buffers, including the trailing null. It is defined to be 256 characters.

6.2.1 Status Bits

The following bit values are used in the `flags` field of `struct bozo_status`, as defined in Section 6.3.4. They record whether or not the associated bnode process currently has a stored core file, whether the bnode execution was stopped because of an excessive number of errors, and whether the mode bits on server binary directories are incorrect.

Name	*Value*	*Description*
`BOZO_HASCORE`	1	Does this bnode have a stored core file?
`BOZO_ERRORSTOP`	2	Was this bnode execution shut down because of an excessive number of errors (more than 10 in a 10-second period)?
`BOZO_BADDIRACCESS`	3	Are the mode bits on the */usr/afs* directory and its descendants (*etc*, *bin*, *logs*, *backup*, *db*, *local*, *etc/KeyFile*, *etc/UserList*) correctly set?

6.2.2 Bnode Activity Bits

This section describes the legal values for the bit positions within the `flags` field of `struct bnode`, as defined in Section 6.3.8. They specify conditions related to the basic activity of the bnode and of the entities relying on it.

Name	*Value*	*Description*
`BNODE_NEEDTIMEOUT`	0x01	This bnode is utilizing the timeout mechanism for invoking actions on its behalf.
`BNODE_ACTIVE`	0x02	The given bnode is in active service.
`BNODE_WAIT`	0x04	Someone is waiting for a status change in this bnode
`BNODE_DELETE`	0x08	This bnode should be deleted at the earliest convenience.
`BNODE_ERRORSTOP`	0x10	This bnode decommissioned because of an excessive number of errors in its associated UNIX processes.

6.2.3 Bnode States

The constants defined in this section are used as values within the `goal` and `fileGoal` fields within a `struct bnode`. They specify either the current state of the associated bnode, or the anticipated state. In particular, the `fileGoal` field, which is the value stored on disk for the bnode, always represents the desired state of the bnode, whether or not it properly reflects the current state. For this reason, only `BSTAT_SHUTDOWN` and `BSTAT_NORMAL` may be used within the `fileGoal` field. The `goal` field may take on any of these values, and accurately reflects the current status of the bnode.

Name	*Value*	*Description*
`BSTAT_SHUTDOWN`	0	The bnode's execution has been (should be) terminated.
`BSTAT_NORMAL`	1	The bnode is (should be) executing normally.
`BSTAT_SHUTTINGDOWN`	2	The bnode is currently being shutdown; execution has not yet ceased.
`BSTAT_STARTINGUP`	3	The bnode execution is currently being commenced; execution has not yet begun.

6.2.4 Pruning Server Binaries

The *BOZO_Prune()* interface function, fully defined in Section 6.6.6.4, allows a properly-authenticated caller to remove ("prune") old copies of server binaries and core files managed by the *BOS Server*. This section identifies the legal values for the `flags` argument to the above function, specifying exactly what is to be pruned.

Name	*Value*	*Description*
`BOZO_PRUNEOLD`	1	Prune all server binaries with the **.OLD* extension.
`BOZO_PRUNEBAK`	2	Prune all server binaries with the **.BAK* extension.
`BOZO_PRUNECORE`	3	Prune core files.

6.2.5 Flag Bits for `struct bnode_proc`

This section specifies the acceptable bit values for the `flags` field in the `struct bnode_proc` structure, as defined in Section 6.3.9. Basically, they are used to record whether or not the UNIX binary associated with the bnode has ever been run, and if so whether it has ever exited.

Name	*Value*	*Description*
BPROC_STARTED	1	Has the associated UNIX process ever been started up?
BPROC_EXITED	2	Has the associated UNIX process ever exited?

6.3 Structures

This section describes the major exported *BOS Server* data structures of interest to application programmers.

6.3.1 struct bozo_netKTime

This structure is used to communicate time values to and from the *BOS Server*. In particular, the *BOZO_GetRestartTime()* and *BOZO_SetRestartTime()* interface functions, described in Sections 6.6.2.5 and 6.6.2.6 respectively, use parameters declared to be of this type.

Four of the fields in this structure specify the hour, minute, second, and day of the event in question. The first field in the layout serves as a mask, identifying which of the above four fields are to be considered when matching the specified time to a given reference time (most often the current time). The bit values that may be used for the mask field are defined in the *afs/ktime.h* include file. For convenience, their values are reproduced here, including some special cases at the end of the table.

Name	*Value*	*Description*
KTIME_HOUR	0x01	Hour should match.
KTIME_MIN	0x02	Minute should match.
KTIME_SEC	0x04	Second should match.
KTIME_DAY	0x08	Day should match.
KTIME_TIME	0x07	All times should match.
KTIME_NEVER	0x10	Special case: never matches.
KTIME_NOW	0x20	Special case: right now.

Fields

int mask - A field of bit values used to specify which of the following field are to be used in computing matches.

short hour - The hour, ranging in value from 0 to 23.

short min - The minute, ranging in value from 0 to 59.

short sec - The second, ranging in value from 0 to 59.

short day - Zero specifies Sunday, other days follow in order.

6.3.2 struct bozo_key

This structure defines the format of an AFS encryption key, as stored in the key file located at */usr/afs/etc/KeyFile* at the host on which the *BOS Server* runs. It is used in the argument list of the *BOZO_ListKeys()* and *BOZO_AddKeys()* interface functions, as described in Sections 6.6.4.4 and 6.6.4.5 respectively.

Fields

char data[8] - The array of 8 characters representing an encryption key.

6.3.3 struct bozo_keyInfo

This structure defines the information kept regarding a given AFS encryption key, as represented by a variable of type `struct bozo_key`, as described in Section 6.3.2 above. A parameter of this type is used by the *BOZO_ListKeys()* function (described in Section 6.6.4.4). It contains fields holding the associated key's modification time, a checksum on the key, and an unused longword field. Note that the `mod_sec` time field listed below is a standard UNIX time value.

Fields

long mod_sec - The time in seconds when the associated key was last modified.

long mod_usec - The number of microseconds elapsed since the second reported in the `mod_sec` field. This field is never set by the *BOS Server*, and should always contain a zero.

unsigned long keyCheckSum - The 32-bit cryptographic checksum of the associated key. A block of zeros is encrypted, and the first four bytes of the result are placed into this field.

long spare2 - This longword field is currently unused, and is reserved for future use.

6.3.4 struct bozo_status

This structure defines the layout of the information returned by the `status` parameter for the interface function *BOZO_GetInstanceInfo()*, as defined in Section 6.6.2.3. The enclosed fields include such information as the temporary and long-term goals for the process instance, an array of bit values recording status information, start and exit times, and associated error codes and signals.

Fields

long goal - The short-term goal for a process instance. Settings for this field are `BSTAT_SHUTDOWN`, `BSTAT_NORMAL`, `BSTAT_SHUTTINGDOWN`, and `BSTAT_STARTINGUP`. These values are fully defined in Section 6.2.3.

long fileGoal - The long-term goal for a process instance. Accepted settings are restricted to a subset of those used by the `goal` field above, as explained in Section 6.2.3.

long procStartTime - The last time the given process instance was started.

long procStarts - The number of process starts executed on the behalf of the given bnode.

long lastAnyExit - The last time the process instance exited for any reason.

long lastErrorExit - The last time a process exited unexpectedly.

long errorCode - The last exit's return code.

long errorSignal - The last signal terminating the process.

long flags - `BOZO_HASCORE`, `BOZO_ERRORSTOP`, and `BOZO_BADDIRACCESS`. These constants are fully defined in Section 6.2.1.

long spare[] - Eight longword spares, currently unassigned and reserved for future use.

6.3.5 struct bnode_ops

This struture defines the base set of operations that each *BOS Server* bnode type (`struct bnode_type`, see Section 6.3.6 below) must implement. They are called at the appropriate times within the *BOS Server* code via the *BOP_** macros (see Section 6.5 and the individual descriptions therein). They allow each bnode type to define its own behavior in response to its particular needs.

Fields

struct bnode *(*create)() - This function is called whenever a bnode of the given type is created. Typically, this function will create bnode structures peculiar to its own type and initialize the new records. Each type implementation may take a different number of parameters. Note: there is no *BOP_** macro defined for this particular function; it is always called directly.

int (*timeout)() - This function is called whenever a timeout action must be taken for this bnode type. It takes a single argument, namely a pointer to a type-specific bnode structure. The *BOP_TIMEOUT* macro is defined to simplify the construction of a call to this function.

int (*getstat)() - This function is called whenever a caller is attempting to get status information concerning a bnode of the given type. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a pointer to a longword in which the desired status value will be placed. The *BOP_GETSTAT* macro is defined to simplify the construction of a call to this function.

int (*setstat)() - This function is called whenever a caller is attempting to set the status information concerning a bnode of the given type. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a longword from which the new status value is obtained. The *BOP_SETSTAT* macro is defined to simplify the construction of a call to this function.

int (*delete)() - This function is called whenever a bnode of this type is being deleted. It is expected that the proper deallocation and cleanup steps will be performed here. It takes a single argument, a pointer to a type-specific bnode structure. The *BOP_DELETE* macro is defined to simplify the construction of a call to this function.

int (*procexit)() - This function is called whenever the UNIX process implementing the given bnode exits. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a pointer to the `struct bnode_proc` (defined in Section 6.3.9), describing that process in detail. The *BOP_PROCEXIT* macro is defined to simplify the construction of a call to this function.

int (*getstring)() - This function is called whenever the status string for the given bnode must be fetched. It takes three parameters. The first is a pointer to a type-specific bnode structure, the second is a pointer to a character buffer, and the third is a longword specifying the size, in bytes, of the above buffer. The *BOP_GETSTRING* macro is defined to simplify the construction of a call to this function.

int (*getparm)() - This function is called whenever a particular parameter string for the given bnode must be fetched. It takes four parameters. The first is a pointer to a type-specific bnode structure, the second is a longword identifying the index of the desired parameter string, the third is a pointer to a character buffer to receive the parameter string, and the fourth and final argument is a longword specifying the size, in bytes, of the above buffer. The *BOP_GETPARM* macro is defined to simplify the construction of a call to this function.

int (*restartp)() - This function is called whenever the UNIX process implementing the bnode of this type is being restarted. It is expected that the stored process command line will be parsed in preparation for the coming execution. It takes a single argument, a pointer to a type-specific bnode structure from which the command line can be located. The *BOP_RESTARTP* macro is defined to simplify the construction of a call to this function.

int (*hascore)() - This function is called whenever it must be determined if the attached process currently has a stored core file. It takes a single argument, a pointer to a type-specific bnode structure from which the name of the core file may be constructed. The *BOP_HASCORE* macro is defined to simplify the construction of a call to this function.

6.3.6 `struct bnode_type`

This structure encapsulates the defining characteristics for a given bnode type. Bnode types are placed on a singly-linked list within the *BOS Server*, and are identified by a null-terminated character string name. They also contain the function array defined in Section 6.3.5, that implements the behavior of that object type. There are three pre-defined bnode types known to the *BOS Server*. Their names are `simple`, `fs`, and `cron`. It is not currently possible to dynamically define and install new *BOS Server* types.

Fields

struct bnode_type *next - Pointer to the next bnode type definition structure in the list.

char *name - The null-terminated string name by which this bnode type is identified.

bnode_ops *ops - The function array that defines the behavior of this given bnode type.

6.3.7 struct bnode_token

This structure is used internally by the *BOS Server* when parsing the command lines with which it will start up process instances. This structure is made externally visible should more types of bnode types be implemented.

Fields

struct bnode_token *next - The next token structure queued to the list.

char *key - A pointer to the token, or parsed character string, associated with this entry.

6.3.8 struct bnode

This structure defines the essence of a *BOS Server* process instance. It contains such important information as the identifying string name, numbers concerning periodic execution on its behalf, the bnode's type, data on start and error behavior, a reference count used for garbage collection, and a set of flag bits.

Fields

char *name - The null-terminated character string providing the instance name associated with this bnode.

long nextTimeout - The next time this bnode should be awakened. At the specified time, the bnode's `flags` field will be examined to see if `BNODE_NEEDTIMEOUT` is set. If so, its *timeout()* operation will be invoked via the *BOP_TIMEOUT()* macro. This field will then be reset to the current time plus the value kept in the `period` field.

long period - This field specifies the time period between timeout calls. It is only used by processes that need to have periodic activity performed.

long rsTime - The time that the *BOS Server* started counting restarts for this process instance.

long rsCount - The count of the number of restarts since the time recorded in the `rsTime` field.

struct bnode_type *type - The type object defining this bnode's behavior.

struct bnode_ops *ops - This field is a pointer to the function array defining this bnode's basic behavior. Note that this is identical to the value of `type->ops`.

This pointer is duplicated here for convenience. All of the *BOP_** macros, discussed in Section 6.5, reference the bnode's operation array through this pointer.

long procStartTime - The last time this process instance was started (executed).

long procStarts - The number of starts (executions) for this process instance.

long lastAnyExit - The last time this process instance exited for any reason.

long lastErrorExit - The last time this process instance exited unexpectedly.

long errorCode - The last exit return code for this process instance.

long errorSignal - The last signal that terminated this process instance.

char *lastErrorName - The name of the last core file generated.

short refCount - A reference count maintained for this bnode.

short flags - This field contains a set of bit fields that identify additional status information for the given bnode. The meanings of the legal bit values, explained in Section 6.2.2, are: `BOZO_NEEDTIMEOUT`, `BOZO_ACTIVE`, `BOZO_WAIT`, `BOZO_DELETE`, and `BOZO_ERRORSTOP`.

char goal - The current goal for the process instance. It may take on any of the values defined in Section 6.2.3, namely `BSTAT_SHUTDOWN`, `BSTAT_NORMAL`, `BSTAT_SHUTTINGDOWN`, and `BSTAT_STARTINGUP`.

This goal may be changed at will by an authorized caller. Such changes affect the current status of the process instance. See the description of the *BOZO_SetStatus()* and *BOZO_SetTStatus()* interface functions, defined in Sections 6.6.3.1 and 6.6.3.2 respectively, for more details.

char fileGoal - This field is similar to `goal`, but represents the goal stored in the on-file *BOS Server* description of this process instance. As with the `goal` field, see functions the description of the *BOZO_SetStatus()* and *BOZO_SetTStatus()* interface functions defined in Sections 6.6.3.1 and 6.6.3.2 respectively for more details.

6.3.9 `struct bnode_proc`

This structure defines all of the information known about each UNIX process the *BOS Server* is currently managing. It contains a reference to the bnode defining the process, along with the command line to be used to start the process, the optional core file name, the UNIX pid, and such things as a flag field to keep additional state information. The *BOS Server* keeps these records on a global singly-linked list.

Fields

struct bnode_proc *next - A pointer to the *BOS Server*'s next process record.

struct bnode *bnode - A pointer to the bnode creating and defining this UNIX process.

char *comLine - The text of the command line used to start this process.

char *coreName - An optional core file component name for this process.

long pid - The UNIX pid, if successfully created.

long lastExit - This field keeps the last termination code for this process.

long lastSignal - The last signal used to kill this process.

long flags - A set of bits providing additional process state. These bits are fully defined in Section 6.2.5, and are: `BPROC_STARTED` and `BPROC_EXITED`.

6.4 Error Codes

This section covers the set of error codes exported by the *BOS Server*, displaying the printable phrases with which they are associated.

Name	*Value*	*Description*
`BZNOTACTIVE`	(39424L)	process not active.
`BZNOENT`	(39425L)	no such entity.
`BZBUSY`	(39426L)	can't do operation now.
`BZEXISTS`	(39427L)	entity already exists.
`BZNOCREATE`	(39428L)	failed to create entity.
`BZDOM`	(39429L)	index out of range.
`BZACCESS`	(39430L)	you are not authorized for this operation.
`BZSYNTAX`	(39431L)	syntax error in create parameter.
`BZIO`	(39432L)	I/O error.
`BZNET`	(39433L)	network problem.
`BZBADTYPE`	(39434L)	unrecognized bnode type.

6.5 Macros

The *BOS Server* defines a set of macros that are externally visible via the *bnode.h* file. They are used to facilitate the invocation of the members of the `struct bnode_ops` function array, which defines the basic operations for a given bnode type. Invocations appear throughout the *BOS Server* code, wherever bnode type-specific operations are required. Note that the only member of the `struct bnode_ops` function array that does *not* have a corresponding invocation macro defined is *create()*, which is always called directly.

6.5.1 *BOP_TIMEOUT()*

```
#define BOP_TIMEOUT(bnode) \
    ((*(bnode)->ops->timeout)((bnode)))
```

Execute the bnode type-specific actions required when a timeout action must be taken. This macro takes a single argument, namely a pointer to a type-specific bnode structure.

6.5.2 *BOP_GETSTAT()*

```
#define BOP_GETSTAT(bnode, a) \
    ((*(bnode)->ops->getstat)((bnode),(a)))
```

Execute the bnode type-specific actions required when a caller is attempting to get status information concerning the bnode. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a pointer to a longword in which the desired status value will be placed.

6.5.3 *BOP_SETSTAT()*

```
#define BOP_SETSTAT(bnode, a) \
    ((*(bnode)->ops->setstat)((bnode),(a)))
```

Execute the bnode type-specific actions required when a caller is attempting to set the status information concerning the bnode. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a longword from which the new status value is obtained.

6.5.4 *BOP_DELETE()*

```
#define BOP_DELETE(bnode) \
    ((*(bnode)->ops->delete)((bnode)))
```

Execute the bnode type-specific actions required when a bnode is deleted. This macro takes a single argument, namely a pointer to a type-specific bnode structure.

6.5.5 *BOP_PROCEXIT()*

```
#define BOP_PROCEXIT(bnode, a) \
    ((*(bnode)->ops->procexit)((bnode),(a)))
```

Execute the bnode type-specific actions required whenever the `UNIX` process implementing the given bnode exits. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a pointer to the `struct bnode_proc` (defined in Section 6.3.9), describing that process in detail.

6.5.6 *BOP_GETSTRING()*

```
#define BOP_GETSTRING(bnode, a, b) \
    ((*(bnode)->ops->getstring)((bnode),(a), (b)))
```

Execute the bnode type-specific actions required when the status string for the given bnode must be fetched. It takes three parameters. The first is a pointer to a type-specific bnode structure, the second is a pointer to a character buffer, and the third is a longword specifying the size, in bytes, of the above buffer.

6.5.7 *BOP_GETPARM()*

```
#define BOP_GETPARM(bnode, n, b, l) \
    ((*(bnode)->ops->getparm)((bnode),(n),(b),(l)))
```

Execute the bnode type-specific actions required when a particular parameter string for the given bnode must be fetched. It takes four parameters. The first is a pointer to a type-specific bnode structure, the second is a longword identifying the index of the desired parameter string, the third is a pointer to a character buffer to receive the parameter string, and the fourth and final argument is a longword specifying the size, in bytes, of the above buffer.

6.5.8 *BOP_RESTARTP()*

```
#define BOP_RESTARTP(bnode) \
    ((*(bnode)->ops->restartp)((bnode)))
```

Execute the bnode type-specific actions required when the UNIX process implementing the bnode of this type is restarted. It is expected that the stored process command line will be parsed in preparation for the coming execution. It takes a single argument, a pointer to a type-specific bnode structure from which the command line can be located.

6.5.9 *BOP_HASCORE()*

```
#define BOP_HASCORE(bnode)  ((*(bnode)->ops->hascore)((bnode)))
```

Execute the bnode type-specific actions required when it must be determined whether or not the attached process currently has a stored core file. It takes a single argument, a pointer to a type-specific bnode structure from which the name of the core file may be constructed.

6.6 Functions

This section covers the *BOS Server* RPC interface routines. They are generated from the *bosint.xg Rxgen* file. At a high level, these functions may be seen as belonging to seven basic classes:

- Creating and removing process entries
- Examining process information
- Starting, stopping, and restarting processes
- Security configuration

- Cell configuration

- Installing/uninstalling server binaries

- Executing commands at the server

The following is a summary of the interface functions and their purpose, divided according to the above classifications:

Creating & Removing Process Entries	
Function Name	**Description**
BOZO_CreateBnode()	Create a process instance.
BOZO_DeleteBnode()	Delete a process instance.

Examining Process Information	
Function Name	**Description**
BOZO_GetStatus()	Get status information for the given process instance.
BOZO_EnumerateInstance()	Get instance name from the *i'th* bnode.
BOZO_GetInstanceInfo()	Get information on the given process instance.
BOZO_GetInstanceParm()	Get text of command line associated with the given process instance.
BOZO_GetRestartTime()	Get one of the *BOS Server* restart times.
BOZO_SetRestartTime()	Set one of the *BOS Server* restart times.
BOZO_GetDates()	Get the modification times for versions of a server binary file.
StartBOZO_GetLog()	Pass the `IN` params when fetching a *BOS Server* log file.
EndBOZO_GetLog()	Get the `OUT` params when fetching a *BOS Server* log file.
BOZO_GetInstanceStrings()	Get strings related to a given process instance.

Starting, Stopping & Restarting Processes	
Function Name	**Description**
BOZO_SetStatus()	Set process instance status and goal.
BOZO_SetTStatus()	Temporarily set process instance status and goal.
BOZO_StartupAll()	Start all existing process instances.
BOZO_ShutdownAll()	Shut down all process instances.
BOZO_RestartAll()	Shut down, then restart all process instances.
BOZO_ReBozo()	Shut down, then restart all process instances and the *BOS Server* itself.
BOZO_Restart()	Restart a given process instance.
BOZO_WaitAll()	Wait until all process instances have reached their goals.

Security Configuration	
Function Name	**Description**
BOZO_AddSUser()	Add a user to the *UserList*.
BOZO_DeleteSUser()	Delete a user from the *UserList*.
BOZO_ListSUsers()	Get the name of the user in the given position in the *UserList* file.
BOZO_ListKeys()	List info about the key at a given index in the key file.
BOZO_AddKey()	Add a key to the key file.
BOZO_DeleteKey()	Delete the entry for an AFS key.
BOZO_SetNoAuthFlag()	Enable or disable authenticated call requirements.

Cell Configuration	
Function Name	**Description**
BOZO_GetCellName()	Get the name of the cell to which the *BOS Server* belongs.
BOZO_SetCellName()	Set the name of the cell to which the *BOS Server* belongs.
BOZO_GetCellHost()	Get the name of a database host given its index.
BOZO_AddCellHost()	Add an entry to the list of database server hosts.
BOZO_DeleteCellHost()	Delete an entry from the list of database server hosts.

Installing/Uninstalling Server Binaries	
Function Name	**Description**
StartBOZO_Install()	Pass the `IN` params when installing a server binary.
EndBOZO_Install()	Get the `OUT` params when installing a server binary.
BOZO_UnInstall()	Roll back from a server binary installation.
BOZO_Prune()	Throw away old versions of server binaries and core files.

Executing Commands at the Server	
Function Name	**Description**
BOZO_Exec()	Execute a shell command at the server.

All of the string parameters in these functions are expected to point to character buffers that are at least `BOZO_BSSIZE` long.

6.6.1 Creating and Removing Processes

The two interface routines defined in this section are used for creating and deleting bnodes, thus determining which processe instances the *BOS Server* must manage.

6.6.1.1 BOZO_CreateBnode — Create a process instance

```
int BOZO_CreateBnode(IN struct rx_connection *z_conn,
                     IN char *type,
                     IN char *instance,
                     IN char *p1,
                     IN char *p2,
                     IN char *p3,
                     IN char *p4,
                     IN char *p5,
                     IN char *p6)
```

Description

This interface function allows the caller to create a bnode (process instance) on the server machine executing the routine.

The instance's type is declared to be the string referenced in the `type` argument. There are three supported instance type names, namely `simple`, `fs`, and `cron` (see Section 5.1 for a detailed examination of the types of bnodes available).

The bnode's name is specified via the `instance` parameter. Any name may be chosen for a *BOS Server* instance. However, it is advisable to choose a name related to the name of the actual binary being instantiated. There are eight well-known names already in common use, corresponding to the ASF system agents. They are as follows:

- *kaserver* for the *Authentication Server*.
- *runntp* for the Network Time Protocol Daemon (*ntpd*).
- *ptserver* for the *Protection Server*.
- *upclient* for the client portion of the *UpdateServer*, which brings over binary files from */usr/afs/bin* directory and configuration files from */usr/afs/etc* directory on the system control machine.
- *upclientbin* for the client portion of the *UpdateServer*, which uses the */usr/afs/bin* directory on the binary distribution machine for this platform's CPU/operating system type.

- *upclientetc* for the client portion of the *UpdateServer*, which references the */usr/afs/etc* directory on the system control machine.
- *upserver* for the server portion of the *UpdateServer*.
- *vlserver* for the *Volume Location Server*.

Up to six command-line strings may be communicated in this routine, residing in arguments `p1` through `p6`. Different types of bnodes allow for different numbers of actual server processes to be started, and the command lines required for such instantiation are passed in this manner.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to create new instances. If successfully created, the new *BOS Server* instance will be appended to the *BosConfig* file kept on the machine's local disk. The *UserList* and *BosConfig* files are examined in detail in Sections 5.3.1 and 5.3.4 respectively.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZEXISTS` The given instance already exists.

`BZBADTYPE` Illegal value provided in the `type` parameter.

`BZNOCREATE` Failed to create desired entry.

6.6.1.2 BOZO_DeleteBnode — Delete a process instance

```
int BOZO_DeleteBnode(IN struct rx_connection *z_conn,
                     IN char *instance)
```

Description

This routine deletes the *BOS Server* bnode whose name is specified by the `instance` parameter. If an instance with that name does not exist, this operation fails. Similarly, if the process or processes associated with the given bnode have not been shut down (see the descriptions for the *BOZO_ShutdownAll()* and *BOZO_ShutdownAll()* interface functions), the operation also fails.

The given bnode's *setstat()* and *delete()* routines from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* and *BOP_DELETE()* macros.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to delete existing instances. If successfully deleted, the old *BOS Server* instance will be removed from the *BosConfig* file kept on the machine's local disk.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZNOENT` The given `instance` name not registered with the *BOS Server*.

`BZBUSY` The process(es) associated with the given `instance` are still active (i.e., a shutdown has not yet been performed or has not yet completed).

6.6.2 Examining Process Information

This section describes the ten interface functions that collectively allow callers to obtain and modify the information stored by the *BOS Server* to describe the set of process that it manages. Among the operations supported by the functions examined here are getting and setting status information, obtaining the instance parameters, times, and dates, and getting the text of log files on the server machine

6.6.2.1 BOZO_GetStatus — Get status information for the given process instance

```
int BOZO_GetStatus(IN struct rx_connection *z_conn,
                   IN char *instance,
                   OUT long *intStat,
                   OUT char **statdescr)
```

Description

This interface function looks up the bnode for the given process `instance` and places its numerical status indicator into `intStat` and its status string (if any) into a buffer referenced by `statdescr`.

The set of values that may be returned in the `intStat` argument are defined fully in Section 6.2.3. Briefly, they are `BSTAT_STARTINGUP`, `BSTAT_NORMAL`, `BSTAT_SHUTTINGDOWN`, and `BSTAT_SHUTDOWN`.

A buffer holding `BOZO_BSSIZE` (256) characters is allocated, and `statdescr` is set to point to it. Not all bnodes types implement status strings, which are used to provide additional status information for the class. An example of one bnode type that does define these strings is `fs`, which exports the following status strings:

- `"file server running"`
- `"file server up; volser down"`
- `"salvaging file system"`
- `"starting file server"`
- `"file server shutting down"`
- `"salvager shutting down"`
- `"file server shut down"`

The given bnode's *getstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* macro.

Error Codes

`BZNOENT` The given process `instance` is not registered with the *BOS Server*.

6.6.2.2 BOZO_EnumerateInstance — Get instance name from *i'th* bnode

```
int BOZO_EnumerateInstance(IN struct rx_connection *z_conn,
                           IN long instance,
                           OUT char **iname);
```

Description

This routine will find the bnode describing process instance number `instance` and return that instance's name in the buffer to which the `iname` parameter points. This function is meant to be used to enumerate all process instances at a *BOS Server*. The first legal `instance` number value is zero, which will return the instance name from the first registered bnode. Successive values for `instance` will return information from successive bnodes. When all bnodes have been thus enumerated, the *BOZO_EnumerateInstance()* function will return `BZDOM`, indicating that the list of bnodes has been exhausted.

Error Codes

`BZDOM` The instance number indicated in the `instance` parameter does not exist.

6.6.2.3 **BOZO_GetInstanceInfo** — Get information on the given process instance

```
int BOZO_GetInstanceInfo(IN struct rx_connection *z_conn,
                         IN char *instance,
                         OUT char **type,
                         OUT struct bozo_status *status)
```

Description

Given the string name of a *BOS Server* `instance`, this interface function returns the `type` of the instance and its associated `status` descriptor. The set of values that may be placed into the `type` parameter are `simple`, `fs`, and `cron` (see Section 5.1 for a detailed examination of the types of bnodes available). The `status` structure filled in by the call includes such information as the goal and file goals, the process start time, the number of times the process has started, exit information, and whether or not the process has a core file.

Error Codes

`BZNOENT` The given process `instance` is not registered with the *BOS Server*.

6.6.2.4 BOZO_GetInstanceParm — Get text of command line associated with the given process instance

```
int BOZO_GetInstanceParm(IN struct rx_connection *z_conn,
                         IN char *instance,
                         IN long num,
                         OUT char **parm)
```

Description

Given the string name of a *BOS Server* process `instance` and an index identifying the associated command line of interest, this routine returns the text of the desired command line. The first associated command line text for the `instance` may be acquired by setting the index parameter, `num`, to zero. If an index is specified for which there is no matching command line stored in the bnode, then the function returns `BZDOM`.

Error Codes

`BZNOENT` The given process `instance` is not registered with the *BOS Server*.

`BZDOM` There is no command line text associated with index `num` for this bnode.

6.6.2.5 BOZO_GetRestartTime — Get one of the *BOS Server* restart times

```
int BOZO_GetRestartTime(IN struct rx_connection *z_conn,
                        IN long type,
                        OUT struct bozo_netKTime *restartTime)
```

Description

The *BOS Server* maintains two different restart times, for itself and all server processes it manages, as described in Section 5.4. Given which one of the two types of restart time is desired, this routine fetches the information from the *BOS Server*. The `type` argument is used to specify the exact restart time to fetch. If `type` is set to one (1), then the general restart time for all agents on the machine is fetched. If `type` is set to two (2), then the new-binary restart time is returned. A value other than these two for the `type` parameter results in a return value of `BZDOM`.

Error Codes

`BZDOM` All illegal value was passed in via the `type` parameter.

6.6.2.6 BOZO_SetRestartTime — Set one of the *BOS Server* restart times

```
int BOZO_SetRestartTime(IN struct rx_connection *z_conn,
                        IN long type,
                        IN struct bozo_netKTime *restartTime)
```

Description

This function is the inverse of the *BOZO_GetRestartTime()* interface routine described in Section 6.6.2.5 above. Given the `type` of restart time and its new value, this routine will set the desired restart time at the *BOS Server* receiving this call. The values for the `type` parameter are identical to those used by *BOZO_GetRestartTime()*, namely one (1) for the general restart time and two (2) for the new-binary restart time.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to set its restart times.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZDOM` All illegal value was passed in via the `type` parameter.

6.6.2.7 BOZO_GetDates — Get the modification times for versions of a server binary file

```
int BOZO_GetDates(IN struct rx_connection *z_conn,
                  IN char *path,
                  OUT long *newtime,
                  OUT long *baktime,
                  OUT long *oldtime)
```

Description

Given a fully-qualified pathname identifying the particular server binary to examine in the `path` argument, this interface routine returns the modification time of that file, along with the modification times for the intermediate (`.BAK`) and old (`.OLD`) versions. The above-mentioned times are deposited into the `newtime`, `baktime` and `oldtime` arguments. Any one or all of the reported times may be set to zero, indicating that the associated file does not exist.

Error Codes

`---` None.

6.6.2.8 StartBOZO_GetLog — Pass the IN params when fetching a *BOS Server* log file

```
int BOZO_StartGetLog(IN struct rx_connection *z_conn,
                     IN char *name)
```

Description

The *BOZO_GetLog()* function defined in the *BOS Server Rxgen* interface file is used to acquire the contents of the given log file from the machine processing the call. It is defined to be a *streamed* function, namely one that can return an arbitrary amount of data. For full details on the definition and use of streamed functions, please refer to the *Streamed Function Calls* section in [4].

This function is created by *Rxgen* in response to the *BOZO_GetLog()* interface definition in the *bosint.xg* file. The *StartBOZO_GetLog()* routine handles passing the `IN` parameters of the streamed call to the *BOS Server*. Specifically, the `name` parameter is used to convey the string name of the desired log file. For the purposes of opening the specified files at the machine being contacted, the current working directory for the *BOS Server* is considered to be */usr/afs/logs*. If the caller is included in the locally-maintained *UserList* file, any pathname may be specified in the `name` parameter, and the contents of the given file will be fetched. All other callers must provide a string that does *not* include the slash character, as it might be used to construct an unauthorized request for a file outside the */usr/afs/logs* directory.

Error Codes

`RXGEN_CC_MARSHAL` The transmission of the *GetLog()* `IN` parameters failed. This and all *rxgen* constant definitions are available from the *rxgen_consts.h* include file.

6.6.2.9 EndBOZO_GetLog — Get the `OUT` params when fetching a *BOS Server* log file

`int` **BOZO_EndGetLog**`(IN struct rx_connection *z_conn)`

Description

This function is created by *Rxgen* in response to the *BOZO_GetLog()* interface definition in the *bosint.xg* file. The *EndBOZO_GetLog()* routine handles the recovery of the `OUT parameters` for this interface call (of which there are none). The utility of such functions is often the value they return. In this case, however, *EndBOZO_GetLog()* always returns success. Thus, it is not even necessary to invoke this particular function, as it is basically a no-op.

Error Codes

`---` Always returns successfully.

6.6.2.10 BOZO_GetInstanceStrings — Get strings related to a given process instance

```
int BOZO_GetInstanceStrings(IN struct rx_connection *z_conn,
                            IN char *instance,
                            OUT char **errorName,
                            OUT char **spare1,
                            OUT char **spare2,
                            OUT char **spare3)
```

Description

This interface function takes the string name of a *BOS Server* instance and returns a set of strings associated with it. At the current time, there is only one string of interest returned by this routine. Specifically, the `errorName` parameter is set to the error string associated with the bnode, if any. The other arguments, `spare1` through `spare3`, are set to the null string. Note that memory is allocated for all of the `OUT` parameters, so the caller should be careful to free them once it is done.

Error Codes

`BZNOENT` The given process `instance` is not registered with the *BOS Server*.

6.6.3 Starting, Stopping, and Restarting Processes

The eight interface functions described in this section allow *BOS Server* clients to manipulate the execution of the process instances the *BOS Server* controls.

6.6.3.1 BOZO_SetStatus — Set process instance status and goal

```
int BOZO_SetStatus(IN struct rx_connection *z_conn,
                   IN char *instance,
                   IN long status)
```

Description

This routine sets the actual status field, as well as the "file goal", of the given `instance` to the value supplied in the `status` parameter. Legal values for `status` are taken from the set described in Section 6.2.3, specifically `BSTAT_NORMAL` and `BSTAT_SHUTDOWN`. For more information about these constants (and about goals/file goals), please refer to Section 6.2.3.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to perform this operation. If successfully modified, the *BOS Server* bnode defining the given instance will be written out to the *BosConfig* file kept on the machine's local disk.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZNOENT` The given `instance` name not registered with the *BOS Server*.

6.6.3.2 BOZO_SetTStatus — Temporarily set process instance status and goal

```
int BOZO_SetTStatus(IN struct rx_connection *z_conn,
                    IN char *instance,
                    IN long status)
```

Description

This interface routine is much like the *BOZO_SetStatus()*, defined in Section 6.6.3.1 above, except that its effect is to set the instance status on a temporary basis. Specifically, the status field is set to the given `status` value, but the "file goal" field is not changed. Thus, the instance's stated goal has not changed, just its current status.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to perform this operation. If successfully modified, the *BOS Server* bnode defining the given instance will be written out to the *BosConfig* file kept on the machine's local disk.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZNOENT` The given `instance` name not registered with the *BOS Server*.

6.6.3.3 BOZO_StartupAll — Start all existing process instances

int **BOZO_StartupAll**(IN struct rx_connection *z_conn)

Description

This interface function examines all bnodes and attempts to restart all of those that have not been explicitly been marked with the BSTAT_SHUTDOWN file goal. Specifically, *BOP_SETSTAT()* is invoked, causing the *setstat()* routine from each bnode's *ops* array to be called. The bnode's flags field is left with the BNODE_ERRORSTOP bit turned off after this call.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to start up bnode process instances.

Error Codes

BZACCESS The caller is not authorized to perform this operation.

6.6.3.4 **BOZO_ShutdownAll** — Shut down all process instances

`int` **BOZO_ShutdownAll**`(IN struct rx_connection *z_conn)`

Description

This interface function iterates through all bnodes and attempts to shut them all down. Specifically, the *BOP_SETSTAT()* is invoked, causing the *setstat()* routine from each bnode's *ops* array to be called, setting that bnode's `goal` field to `BSTAT_SHUTDOWN`.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to perform this operation.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

6.6.3.5 BOZO_RestartAll — Shut down, then restart all process instances

int **BOZO_RestartAll**(IN struct rx_connection *z_conn)

Description

This interface function shuts down every *BOS Server* process instance, waits until the shutdown is complete (i.e., all instances are registered as being in state BSTAT_SHUTDOWN), and then starts them all up again. While all the processes known to the *BOS Server* are thus restarted, the invocation of the *BOS Server* itself does *not* share this fate. For simulation of a truly complete machine restart, as is necessary when an far-reaching change to a database file has been made, use the *BOZO_ReBozo()* interface routine defined in Section 6.6.3.6 below.

The given bnode's *getstat()* and *setstat()* routines from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* and *BOP_SETSTAT()* macros.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to restart bnode process instances.

Error Codes

BZACCESS The caller is not authorized to perform this operation.

6.6.3.6 BOZO_ReBozo — Shut down, then restart all process instances and the *BOS Server* itself

```
int BOZO_ReBozo(IN struct rx_connection *z_conn)
```

Description

This interface routine is identical to the *BOZO_RestartAll()* call, defined in Section 6.6.3.5 above, except for the fact that the *BOS Server* itself is restarted in addition to all the known bnodes. All of the *BOS Server*'s open file descriptors are closed, and the standard *BOS Server* binary image is started via *execve()*.

The given bnode's *getstat()* and *setstat()* routines from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* and *BOP_SETSTAT()* macros.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to restart bnode process instances.

Error Codes

BZACCESS	The caller is not authorized to perform this operation.

6.6.3.7 **BOZO_Restart** — Restart a given process instance

```
int BOZO_Restart(IN struct rx_connection *z_conn,
                 IN char *instance)
```

Description

This interface function is used to shut down and then restart the process instance identified by the `instance` string passed as an argument.

The given bnode's *getstat()* and *setstat()* routines from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* and *BOP_SETSTAT()* macros.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to restart bnode process instances.

Error Codes

- `BZACCESS` The caller is not authorized to perform this operation.
- `BZNOENT` The given `instance` name not registered with the *BOS Server*.

6.6.3.8 BOZO_WaitAll — Wait until all process instances have reached their goals

`int` **BOZO_WaitAll**`(IN struct rx_connection *z_conn)`

Description

This interface function is used to synchronize with the status of the bnodes managed by the *BOS Server*. Specifically, the *BOZO_WaitAll()* call returns when each bnode's current status matches the value in its short-term `goal` field. For each bnode it manages, the *BOS Server* thread handling this call invokes the *BOP_GETSTAT()* macro, waiting until the bnode's status and goals line up.

Typically, the *BOZO_WaitAll()* routine is used to allow a program to wait until all bnodes have terminated their execution (i.e., all `goal` fields have been set to `BSTAT_SHUTDOWN` and all corresponding processes have been killed). Note, however, that this routine may also be used to wait until all bnodes *start up*. The true utility of this application of *BOZO_WaitAll()* is more questionable, since it will return when all bnodes have simply *commenced execution*, which does not imply that they have completed their initialization phases and are thus rendering their normal services.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to wait on bnodes through this interface function.

The given bnode's *getstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* macro.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

6.6.4 Security Configuration

This section describes the seven *BOS Server* interface functions that allow a properly-authorized person to examine and modify certain data relating to system security. Specifically, it allows for manipulation of the list of adminstratively "privileged" individuals, the set of Kerberos keys used for file service, and whether authenticated connections should be required by the *BOS Server* and all other AFS server agents running on the machine.

6.6.4.1 **BOZO_AddSUser** — Add a user to the *UserList*

```
int BOZO_AddSUser(IN struct rx_connection *z_conn,
                  IN char *name);
```

Description

This interface function is used to add the given user `name` to the *UserList* file of priviledged *BOS Server* principals. Only individuals already appearing in the *UserList* are permitted to add new entries. If the given user `name` already appears in the file, the function fails. Otherwise, the file is opened in append mode and the `name` is written at the end with a trailing newline.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`EEXIST` The individual specified by `name` is already on the *UserList*.

`EIO` If the *UserList* file could not be opened or closed.

6.6.4.2 BOZO_DeleteSUser — Delete a user from the *UserList*

```
int BOZO_DeleteSUser(IN struct rx_connection *z_conn,
                     IN char *name)
```

Description

This interface function is used to delete the given user `name` from the *UserList* file of priviledged *BOS Server* principals. Only individuals already appearing in the *UserList* are permitted to delete existing entries. The file is opened in read mode, and a new file named *UserList.NXX* is created in the same directory and opened in write mode. The original *UserList* is scanned, with each entry copied to the new file if it doesn't match the given `name`. After the scan is done, all files are closed, and the *UserList.NXX* file is renamed to *UserList*, overwriting the original.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`-1` The *UserList* file could not be opened.

`EIO` The *UserList.NXX* file could not be opened, or an error occured in the file close operations.

`ENOENT` The given `name` was not found in the original *UserList* file.

6.6.4.3 BOZO_ListSUsers — Get the name of the user in the given position in the *UserList* file

```
int BOZO_ListSUsers(IN struct rx_connection *z_conn,
                    IN long an,
                    OUT char **name)
```

Description

This interface function is used to request the `name` of priviledged user in the `an`'th slot in the *BOS Server*'s *UserList* file. The string placed into the `name` parameter may be up to 256 characters long, including the trailing null.

Error Codes

`1` The *UserList* file could not be opened, or an invalid value was specified for `an`.

6.6.4.4 BOZO_ListKeys — List info about the key at a given index in the key file

```
int BOZO_ListKeys(IN struct rx_connection *z_conn,
                  IN long an,
                  OUT long *kvno,
                  OUT struct bozo_key *key,
                  OUT struct bozo_keyInfo *keyinfo)
```

Description

This interface function allows its callers to specify the index of the desired AFS encryption key, and to fetch information regarding that key. If the caller is properly authorized, the version number of the specified key is placed into the `kvno` parameter. Similarly, a description of the given key is placed into the `keyinfo` parameter. When the *BOS Server* is running in noauth mode, the key itself will be copied into the `key` argument, otherwise the `key` structure will be zeroed. The data placed into the `keyinfo` argument, declared as a `struct bozo_keyInfo` as defined in Section 6.3.3, is obtained as follows. The `mod_sec` field is taken from the value of `st_mtime` after *stat()*ing */usr/afs/etc/KeyFile*, and the `mod_usec` field is zeroed. The *keyCheckSum* is computed by an *Authentication Server* routine, which calculates a 32-bit cryptographic checksum of the key, encrypting a block of zeros and then using the first 4 bytes as the checksum.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to obtain information regarding the list of AFS keys held by the given *BOS Server*.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZDOM` An invalid index was found in the `an` parameter.

`KABADKEY` Defined in the exported *kautils.h* header file corresponding to the *Authentication Server*, this return value indicates a problem with generating the checksum field of the `keyinfo` parameter.

6.6.4.5 BOZO_AddKey — Add a key to the key file

```
int BOZO_AddKey(IN struct rx_connection *z_conn,
                IN long an,
                IN struct bozo_key *key)
```

Description

This interface function allows a properly-authorized caller to set the value of key version number `an` to the given AFS `key`. If a slot is found in the key file */usr/afs/etc/KeyFile* marked as key version number `an`, its value is overwritten with the `key` provided. If an entry for the desired key version number does not exist, the key file is grown, and the new entry filled with the specified information.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to add new entries into the list of AFS keys held by the *BOS Server*.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_FULL` The system key file already contains the maximum number of keys (`AFSCONF_MAXKEYS`, or 8). These two constant defintions are available from the *cellconfig.h* and *keys.h* AFS include files respectively.

6.6.4.6 BOZO_DeleteKey — Delete the entry for an AFS key

```
int BOZO_DeleteKey(IN struct rx_connection *z_conn,
                   IN long an)
```

Description

This interface function allows a properly-authorized caller to delete key version number `an` from the key file, */usr/afs/etc/KeyFile*. The existing keys are scanned, and if one with key version number `an` is found, it is removed. Any keys occurring after the deleted one are shifted to remove the file entry entirely.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to delete entries from the list of AFS keys held by the *BOS Server*.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_NOTFOUND` An entry for key version number `an` was not found. This constant defintion is available from the *cellconfig.h* AFS include file.

6.6.4.7 BOZO_SetNoAuthFlag — Enable or disable requirement for authenticated calls

```
int BOZO_SetNoAuthFlag(IN struct rx_connection *z_conn,
                       IN long flag)
```

Description

This interface routine controls the level of authentication imposed on the *BOS Server* and all other AFS server agents on the machine by manipulating the *NoAuth* file in the */usr/afs/local* directory on the server. If the `flag` parameter is set to zero (0), the *NoAuth* file will be removed, instructing the *BOS Server* and AFS agents to authenenticate the RPCs they receive. Otherwise, the file is created as an indication to honor all RPC calls to the *BOS Server* and AFS agents, regardless of the credentials carried by callers.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

6.6.5 Cell Configuration

The five interface functions covered in this section all have to do with manipulating the configuration information of the machine on which the *BOS Server* runs. In particular, one may get and set the cell name for that server machine, enumerate the list of server machines running database servers for the cell, and add and delete machines from this list.

6.6.5.1 BOZO_GetCellName — Get the name of the cell to which the *BOS Server* belongs

```
int BOZO_GetCellName(IN struct rx_connection *z_conn,
                     OUT char **name)
```

Description

This interface routine returns the name of the cell to which the given *BOS Server* belongs. The *BOS Server* consults a file on its local disk, */usr/afs/etc/ThisCell* to obtain this information. If this file does not exist, then the *BOS Server* will return a null string.

Error Codes

`AFSCONF_UNKNOWN` The *BOS Server* could not access the cell name file. This constant defintion is available from the *cellconfig.h* AFS include file.

6.6.5.2 BOZO_SetCellName — Set the name of the cell to which the *BOS Server* belongs

```
int BOZO_SetCellName(IN struct rx_connection *z_conn,
                     IN char *name)
```

Description

This interface function allows the caller to set the name of the cell to which the given *BOS Server* belongs. The *BOS Server* writes this information to a file on its local disk, */usr/afs/etc/ThisCell*. The current contents of this file are first obtained, along with other information about the current cell. If this operation fails, then *BOZO_SetCellName()* also fails. The string `name` provided as an argument is then stored in *ThisCell*.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to set the name of the cell to which the machine executing the given *BOS Server* belongs.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_NOTFOUND` Information about the current cell could not be obtained. This constant definition, along with `AFSCONF_FAILURE` appearing below, is availabel from the *cellconfig.h* AFS include file.

`AFSCONF_FAILURE` New cell name could not be written to file.

6.6.5.3 BOZO_GetCellHost — Get the name of a database host given its index

```
int BOZO_GetCellHost(IN struct rx_connection *z_conn,
                     IN long awhich,
                     OUT char **name)
```

Description

This interface routine allows the caller to get the name of the host appearing in position `awhich` in the list of hosts acting as database servers for the *BOS Server*'s cell. The first valid position in the list is index zero. The host's name is deposited in the character buffer pointed to by `name`. If the value of the index provided in `awhich` is out of range, the function fails and a null string is placed in `name`.

Error Codes

`BZDOM` The host index in `awhich` is out of range.

`AFSCONF_NOTFOUND` Information about the current cell could not be obtained. This constant defintion may be found in the *cellconfig.h* AFS include file.

6.6.5.4 **BOZO_AddCellHost** — Add an entry to the list of database server hosts

```
int BOZO_AddCellHost(IN struct rx_connection *z_conn,
                     IN char *name)
```

Description

This interface function allows properly-authorized callers to add a `name` to the list of hosts running AFS database server processes for the *BOS Server*'s home cell. If the given name does not already appear in the database server list, a new entry will be created. Regardless, the mapping from the given name to its IP address will be recomputed, and the cell database file, */usr/afs/etc/CellServDB* will be updated.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to add an entry to the list of host names providing database services for the *BOS Server*'s home cell.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_NOTFOUND` Information about the current cell could not be obtained. This constant defintion may be found in the *cellconfig.h* AFS include file.

6.6.5.5 BOZO_DeleteCellHost — Delete an entry from the list of database server hosts

```
int BOZO_DeleteCellHost(IN struct rx_connection *z_conn,
                        IN char *name)
```

Description

This interface routine allows properly-authorized callers to remove a given `name` from the list of hosts running AFS database server processes for the *BOS Server*'s home cell. If the given name does not appear in the database server list, this function will fail. Otherwise, the matching entry will be removed, and the cell database file, */usr/afs/etc/CellServDB* will be updated.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to delete an entry from the list of host names providing database services for the *BOS Server*'s home cell.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_NOTFOUND` Information about the current cell could not be obtained. This constant defintion may be found in the *cellconfig.h* AFS include file.

6.6.6 Installing/Uninstalling Server Binaries

There are four *BOS Server* interface routines that allow administrators to install new server binaries and to roll back to older, perhaps more reliable, executables. They also allow for stored images of the old binaries (as well as core files) to be "pruned", or selectively deleted.

6.6.6.1 StartBOZO_Install — Pass the `IN` params when installing a server binary

```
int StartBOZO_Install(IN struct rx_connection *z_conn,
                      IN char *path,
                      IN long size,
                      IN long flags,
                      IN long date)
```

Description

The *BOZO_Install()* function defined in the *BOS Server Rxgen* interface file is used to deliver the executable image of an AFS server process to the given server machine and then installing it in the appropriate directory there. It is defined to be a *streamed* function, namely one that can deliver an arbitrary amount of data. For full details on the definition and use of streamed functions, please refer to the *Streamed Function Calls* section in [4].

This function is created by *Rxgen* in response to the *BOZO_Install()* interface definition in the *bosint.xg* file. The *StartBOZO_Install()* routine handles passing the `IN` parameters of the streamed call to the *BOS Server*. Specifically, the `apath` argument specifies the name of the server binary to be installed (including the full pathname prefix, if necessary). Also, the length of the binary is communicated via the `size` argument, and the modification time the caller wants the given file to carry is placed in `date`. The `flags` argument is currently ignored by the *BOS Server*.

After the above parameters are delivered with *StartBOZO_Install()*, the *BOS Server* creates a file with the name given in the `path` parameter followed by a *.NEW* postfix. The `size` bytes comprising the text of the executable in question are then read over the RPC channel and stuffed into this new file. When the transfer is complete, the file is closed. The existing versions of the server binary are then "demoted"; the **.BAK* version (if it exists) is renamed to **.OLD*. overwriting the existing **.OLD* version if and only if an **.OLD* version does not exist, or if a **.OLD* exists and the .BAK file is at least seven days old. The main binary is then renamed to **.BAK*. Finally, the **.NEW* file is renamed to be the new standard binary image to run, and its modification time is set to `date`.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to install server software onto the machine on which the *BOS Server* runs.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`100` An error was encountered when writing the binary image to the local disk file. The truncated file was closed and deleted on the *BOS Server*.

`101` More than `size` bytes were delivered to the *BOS Server* in the RPC transfer of the executable image.

`102` Fewer than `size` bytes were delivered to the *BOS Server* in the RPC transfer of the executable image.

6.6.6.2 EndBOZO_Install — Get the `OUT` params when installing a server binary

`int` **EndBOZO_Install**`(IN struct rx_connection *z_conn)`

Description

This function is created by *Rxgen* in response to the *BOZO_Install()* interface definition in the *bosint.xg* file. The *EndBOZO_Install()* routine handles the recovery of the `OUT parameters` for this interface call, of which there are none. The utility of such functions is often the value they return. In this case, however, *EndBOZO_Install()* always returns success. Thus, it is not even necessary to invoke this particular function, as it is basically a no-op.

Error Codes

`---` Always returns successfully.

6.6.6.3 BOZO_UnInstall — Roll back from a server binary installation

```
int BOZO_UnInstall(IN struct rx_connection *z_conn,
                   IN char *path)
```

Description

This interface function allows a properly-authorized caller to "roll back" from the installation of a server binary. If the **.BAK* version of the server named `path` exists, it will be renamed to be the main executable file. In this case, the **.OLD* version, if it exists, will be renamed to **.BAK*. If a **.BAK* version of the binary in question is not found, the **.OLD* version is renamed as the new standard binary file. If neither a **.BAK* or a **.OLD* version of the executable can be found, the function fails, returning the low-level `UNIX` error generated at the server.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to roll back server software on the machine on which the *BOS Server* runs.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

6.6.6.4 **BOZO_Prune** — Throw away old versions of server binaries and core files

```
int BOZO_Prune(IN struct rx_connection *z_conn,
               IN long flags)
```

Description

This interface routine allows a properly-authorized caller to prune the saved versions of server binaries resident on the machine on which the *BOS Server* runs. The */usr/afs/bin* directory on the server machine is scanned in directory order. If the `BOZO_PRUNEOLD` bit is set in the `flags` argument, every file with the **.OLD* extension is deleted. If the `BOZO_PRUNEBAK` bit is set in the `flags` argument, every file with the **.BAK* extension is deleted. Next, the */usr/afs/logs* directory is scanned in directory order. If the `BOZO_PRUNECORE` bit is set in the `flags` argument, every file with a name beginning with the prefix *core* is deleted.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to prune server software binary versions and core files on the machine on which the *BOS Server* runs.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

6.6.7 Executing Commands at the Server

There is a single interface function defined by the *BOS Server* that allows execution of arbitrary programs or scripts on any server machine on which a *BOS Server* process is active.

6.6.7.1 **BOZO_Exec** — Execute a shell command at the server

```
int BOZO_Exec(IN struct rx_connection *z_conn,
              IN char *cmd)
```

Description

This interface routine allows a properly-authorized caller to execute any desired shell command on the server on which the given *BOS Server* runs. There is currently no provision made to pipe the output of the given command's execution back to the caller through the RPC channel.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to execute arbitrary shell commands on the server machine on which the *BOS Server* runs via this call.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

Bibliography

[1] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[2] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[3] Zayas, Edward R., Transarc Corporation. *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*, FS-00-D164, Pittsburgh, PA, April 1991.

[4] Zayas, Edward R., Transarc Corporation. *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*, FS-00-D162, Pittsburgh, PA, April 1991.

[5] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[6] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Ubik - A Library For Managing Ubiquitous Data*, ITCID, Pittsburgh, PA, Month, 1988.

[7] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Quorum Completion*, ITCID, Pittsburgh, PA, Month, 1988.

[8] S. R. Kleinman. *Vnodes: An Architecture for Multiple file System Types in Sun UNIX*, Conference Proceedings, 1986 Summer Usenix Technical Conference, pp. 238-247, El Toro, CA, 1986.

File Server/Cache Manager Interface

Chapter 1

Overview

1.1 Introduction

1.1.1 The AFS 3.1 Distributed File System

AFS 3.1 is a distributed file system (DFS) designed to meet the following set of requirements:

- **Server-client model**: Permanent file storage for AFS is maintained by a collection of *file server* machines. This centralized storage is accessed by individuals running on *client* machines, which also serve as the computational engines for those users. A single machine may act as both an AFS file server and client simultaneously. However, file server machines are generally assumed to be housed in a secure environment, behind locked doors.

- **Scale**: Unlike other existing DFSs, AFS was designed with the specific goal of supporting a very large user community. Unlike the rule-of-thumb ratio of 20 client machines for every server machine (20:1) used by Sun Microsystem's NFS distributed file system [4][5], the AFS architecture aims at smoothly supporting client/server ratios more along the lines of 200:1 within a single installation.

  AFS also provides another, higher-level notion of scalability. Not only can each independently-administered AFS site, or *cell*, grow very large (on the order of tens of thousands of client machines), but individual cells may easily collaborate to form a single, unified file space composed of the union of the individual name spaces. Thus, users have the image of a single UNIX file system tree rooted at the */afs* directory on their machine. Access to files in this tree is performed with the standard UNIX commands, editors, and tools, regardless of a file's location.

These cells and the files they export may be geographically dispersed, thus requiring client machines to access remote file servers across network pathways varying widely in speed, latency, and reliability. The AFS architecture encourages this concept of a single, **wide-area file system**. As of this writing, the community AFS filespace includes sites spanning the continental United States and Hawaii, and also reaches overseas to various installations in Europe, Japan, and Australia.

- **Performance**: This is a critical consideration given the scalability and connectivity requirements described above. A high-performance system in the face of high client/server ratios and the existence of low-bandwidth, high-latency network connections as well as the normal high-speed ones is achieved by two major mechanisms:

  - **Caching**: Client machines make extensive use of caching techniques wherever possible. One important application of this methodology is that each client is required to maintain a cache of files it has accessed from AFS file servers, performing its operations exclusively on these local copies. This file cache is organized in a least-recently-used (LRU) fashion. Thus, each machine will build a local working set of objects being referenced by its users. As long as the cached images remain "current" (i.e., compatible with the central version stored at the file servers), operations may be performed on these files without further communication with the central servers. This results in significant reductions in network traffic and server loads, paving the way for the target client/server ratios.

    This file cache is typically located on the client's local hard disk, although a strictly in-memory cache is also supported. The disk cache has the advantage that its contents will survive crashes and reboots, with the expectation that the majority of cached objects will remain current. The local cache parameters, including the maximum number of blocks it may occupy on the local disk, may be changed on the fly. In order to avoid having the size of the client file cache become a limit on the length of an AFS file, caching is actually performed on **chunks** of the file. These chunks are typically 64 Kbytes in length, although the chunk size used by the client is settable when the client starts up.

  - **Callbacks**: The use of caches by the file system, as described above, raises the thorny issue of cache consistency. Each client must efficiently determine whether its cached file chunks are identical to the corresponding sections of the file as stored at the server machine before allowing a user to operate on those chunks. AFS employs the notion of a **callback** as the backbone of its cache consistency algorithm. When a server machine delivers one or more chunks of a file to a client, it also includes a callback "promise" that the client will be notified if any modifications are made to the data in the file. Thus, as long as the client machine is in possession of a callback for a file, it knows it is correctly synchronized with the centrally-stored version, and allows its users to operate on it as desired without any further interaction with the server. Before a file server stores a more recent version of a file on its own disks, it will first break all outstanding callbacks on this item. A callback will eventually time out, even if there are no changes to the file or directory it covers.

- **Location transparency**: The typical AFS user does not know which server or servers houses any of his or her files. In fact, the user's storage may be distributed among several servers. This location transparency also allows user data to be migrated between servers without users having to take corrective actions, or even becoming aware of the shift.
- **Reliability**: The crash of a server machine in any distributed file system will cause the information it hosts to become unavailable to the user community. The same effect is caused when server and client machines are isolated across a network partition. AFS addresses this situation by allowing data to be **replicated** across two or more servers in a read-only fashion. If the client machine loses contact with a particular server from which it is attempting to fetch data, it hunts among the remaining machines hosting replicas, looking for one that is still in operation. This search is performed without the user's knowledge or intervention, smoothly masking outages whenever possible. Each client machine will automatically perform periodic probes of machines on its list of known servers, updating its internal records concerning their status. Consequently, server machines may enter and exit the pool without administrator intervention.

  Replication also applies to the various databases employed by the AFS server processes. These system databases are read/write replicated with a single synchronization site at any instant. If a synchronization site is lost due to failure, the remaining database sites elect a new synchronization site automatically without operator intervention.
- **Security**: A production file system, especially one which allows and encourages transparent access between administrative domains, must be conscious of security issues. AFS considers the server machines as "trusted", being kept behind locked doors and only directly manipulated by administrators. On the other hand, client machines are, by definition, assumed to exist in inherently insecure environments. These client machines are recognized to be fully accessible to their users, making AFS servers open to attacks mounted by possibly modified hardware, operating systems, and software from its clients.

To provide credible file system security, AFS employs an authentication system based on the Kerberos facility developed by Project Athena at MIT [6][7]. Users operating from client machines are required to interact with *Authentication Server* agents running on the secure server machines to generate secure **tokens** of identity. These tokens express the user's identity in an encrypted fashion, and are stored in the kernel of the client machine. When the user attempts to fetch or store files, the server may *challenge* the user to verify his or her identity. This challenge, hidden from the user and handled entirely by the RPC layer, will transmit this token to the file server involved in the operation. The server machine, upon decoding the token and thus discovering the user's true identity, will allow the caller to perform the operation if permitted.

- **Access control**: The standard UNIX access control mechanism associates *mode bits* with every file and directory, applying them based on the user's numerical identifier and the user's membership in various groups. AFS has augmented this traditional access control mechanism with *Access Control Lists (ACLs)*. Every AFS directory has an associated ACL which defines the *principals* or parties that may operate on all files contained in the directory, and which operations these principals may perform. Rights granted by these ACLs include read, write, delete, lookup, insert (create new files, but don't overwrite old files), and administer (change the ACL). Principals on these ACLs include individual users and groups of users. These groups may be defined by AFS users without administrative intervention. AFS ACLs provide for much finer-grained access control for its files.
- **Administrability**: Any system with the scaling goals of AFS must pay close attention to its ease of administration. The task of running an AFS installation is facilitated via the following mechanisms:
  - **Pervasive RPC interfaces**: Access to AFS server agents is performed mostly via RPC interfaces. Thus, servers may be queried and operated upon regardless of their location. In combination with the security system outlined above, even administrative functions such as instigating backups, reconfiguring server machines, and stopping and restarting servers may be performed by an administrator sitting in front of *any* AFS-capable machine, as long as the administrator holds the proper tokens.
  - **Replication**: As AFS supports read-only replication for user data and read-write replication for system databases, much of the system reconfiguration work in light of failures is performed transparently and without human intervention. Administrators thus typically have more time to respond to many common failure situations.
  - **Data mobility**: Improved and balanced utilization of disk resources is facilitated by the fact that AFS supports transparent relocation of user data

between partitions on a single file server machine or between two different machines. In a situation where a machine must be brought down for an extended period, all its storage may be migrated to other servers so that users may continue their work completely unaffected.

- **Automated "nanny" services**: Each file server machine runs a *BOS Server* process, which assists in the machine's administration. This server is responsible for monitoring the health of the AFS agents under its care, bringing them up in the proper order after a system reboot, answering requests as to their status and restarting them when they fail. It also accepts commands to start, suspend, or resume these processes, and install new server binaries. Accessible via an RPC interface, this supervisory process relieves administrators of some oversight responsibilities and also allows them to perform their duties from any machine running AFS, regardless of location or geographic distance from the targeted file server machine.

- **On-line backup**: Backups may be performed on the data stored by the AFS file server machines without bringing those machines down for the duration. Copy-on-write "snapshots" are taken of the data to be preserved, and tape backup is performed from these clones. One added benefit is that these backup clones are on-line and accessible by users. Thus, if someone accidentally deletes a file that is contained in their last snapshot, they may simply copy its contents as of the time the snapshot was taken back into their active workspace. This facility also serves to improve the administrability of the system, greatly reducing the number of requests to restore data from tape.

- **On-line help**: The set of provided program tools used to interact with the active AFS agents are self-documenting in that they will accept command-line requests for help, displaying descriptive text in response.

- **Statistics**: Each AFS agent facilitates collection of statistical data on its performance, configuration, and status via its RPC interface. Thus, the system is easy to monitor. One tool that takes advantage of this facility is the `scout` program. `Scout` polls file server machines periodically, displaying usage statistics, current disk capacities, and whether the server is unavailable. Administrators monitoring this information can thus quickly react to correct overcrowded disks and machine crashes.

• **Coexistence**: Many organizations currently employ other distributed file systems, most notably NFS. AFS was designed to run simultaneously with other DFSs without interfering in their operation. In fact, an NFS-AFS translator agent exists that allows pure-NFS client machines to transparently access files in the AFS community.

- **Portability**: Because AFS is implemented using the standard **VFS** and **vnode** interfaces pioneered and advanced by Sun Microsystems, AFS is easily portable between different platforms from a single vendor or from different vendors.

1.1.2 Scope of this Document

This document is a member of a documentation suite providing specifications of the operations and interfaces offered by the various AFS servers and agents. Specifically, this document will focus on two of these system agents:

- *File Server*: This AFS entity is responsible for providing a central disk repository for a particular set of files and for making these files accessible to properly-authorized users running on client machines. The *File Server* is implemented as a user-space process
- *Cache Manager*: This code, running within the kernel of an AFS client machine, is a user's representative in communicating with the *File Server*s, fetching files back and forth into the local cache as needed. The *Cache Manager* also keeps information as to the composition of its own cell as well as the other AFS cells in existence. It resolves file references and operations, determining the proper *File Server* (or group of *File Server*s) that may satisfy the request. In addition, it is also a reliable repository for the user's authentication information, holding on to their tokens and wielding them as necessary when challenged.

1.1.3 Related Documents

The full AFS specification suite of documents is listed below:

- *AFS-3 Programmer's Reference: Architectural Overview*: This paper provides an architectual overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and examining their interactions.
- *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.
- *AFS-3 Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for providing two-way mappings between printable user

names and their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users, which are suitable for placement on ACLs.

- *AFS-3 Programmer's Reference: BOS Server Interface*: This paper explicates the "nanny" service described above, which assists in the administrability of the AFS environment.
- *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*: This document specifies the design and operation of the remote procedure call and lightweight process packages used by AFS.

In addition to these papers, the AFS 3.1 product is delivered with its own user, administrator, installation, and command reference documents.

1.2 Basic Concepts

To properly understand AFS operation, specifically the tasks and objectives of the *File Server* and *Cache Manager*, it is necessary to introduce and explain the following concepts:

- **Cell**: A cell is the set of server and client machines operated by an administratively independent organization. The cell administrators make decisions concerning such issues as server deployment and configuration, user backup schedules, and replication strategies on their own hardware and disk storage completely independently from those implemented by other cell administrators regarding their own domains. Every client machine belongs to exactly one cell, and uses that information to determine the set of database servers it uses to locate system resources and generate authentication information.
- **Volume**: AFS disk partitions do not directly host individual user files or directories. Rather, connected subtrees of the system's directory structure are placed into containers called **volumes**. Volumes vary in size dynamically as objects are inserted, overwritten, and deleted. Each volume has an associated **quota**, or maximum permissible storage. A single UNIX disk partition may host one or more volumes, and in fact may host as many volumes as physically fit in the storage space. However, a practical maximum is 3,500 volumes per disk partition, since this is the highest number currently handled by the *salvager* program. The *salvager* is run on occasions where the volume structures on disk are inconsistent, repairing the

damage. A compile-time constant within the *salvager* imposes the above limit, causing it to refuse to repair any inconsistent partition with more than 3,500 volumes.

Volumes serve many purposes within AFS. First, they reduce the number of objects with which an administrator must be concerned, since operations are normally performed on an entire volume at once (and thus on all files and directories contained within the volume). In addition, volumes are the unit of replication, data mobility between servers, and backup. Disk utilization may be balanced by transparently moving volumes between partitions.

- **Mount Point**: The connected subtrees contained within individual volumes stored at AFS file server machines are "glued" to their proper places in the file space defined by a site, forming a single, apparently seamless UNIX tree. These attachment points are referred to as **mount points**. Mount points are persistent objects, implemented as symbolic links whose contents obey a stylized format. Thus, AFS mount points differ from NFS-style *mounts*. In the NFS environment, the user dynamically mounts entire remote disk partitions using any desired name. These mounts do not survive client restarts, and do not insure a uniform namespace between different machines.

  As a *Cache Manager* resolves an AFS pathname as part of a file system operation initiated by a user process, it recognizes mount points and takes special action to resolve them. The *Cache Manager* consults the appropriate *Volume Location Server* to discover the *File Server* (or set of *File Servers*) hosting the indicated volume. This location information is cached, and the *Cache Manager* then proceeds to contact the listed *File Server*(s) in turn until one is found that responds with the contents of the volume's root directory. Once mapped to a real file system object, the pathname resolution proceeds to the next component.

- **Database Server**: A set of AFS databases is required for the proper functioning of the system. Each database may be replicated across two or more file server machines. Access to these databases is mediated by a database server process running at each replication site. One site is declared to be the synchronization site, the sole location accepting requests to modify the databases. All other sites are read-only with respect to the set of AFS users. When the synchronization site receives an update to its database, it immediately distributes it to the other sites. Should a synchronization site go down through either a hard failure or a network partition, the remaining sites will automatically elect a new synchronization site if they form a **quorum**, or majority. This insures that multiple synchronization sites do not become active in the network partition scenario.

  The classes of AFS database servers are listed below:

- *Authentication Server*: This server maintains the authentication database used to generate tokens of identity.
- *Protection Server*: This server maintains mappings between human-readable user account names and their internal numerical AFS identifiers. It also manages the creation, manipulation, and update of user-defined groups suitable for use on ACLs.
- *Volume Location Server*: This server exports information concerning the location of the individual volumes housed within the cell.

1.3 Document Layout

Following this introduction and overview, Chapter 2 describes the architecture of the *File Server* process design. Similarly, Chapter 3 describes the architecture of the in-kernel *Cache Manager* agent. Following these architectural examinations, Chapter 4 provides a set of basic coding definitions common to both the AFS *File Server* and *Cache Manager*, required to properly understand the interface specifications which follow. Chapter 5 then proceeds to specify the various *File Server* interfaces. The myriad *Cache Manager* interfaces are presented in Chapter 6, thus completing the document.

Chapter 2

File Server **Architecture**

2.1 Overview

The AFS *File Server* is a user-level process that presides over the raw disk partitions on which it supports one or more volumes. It provides "half" of the fundamental service of the system, namely exporting and regimenting access to the user data entrusted to it. The *Cache Manager* provides the other half, acting on behalf of its human users to locate and access the files stored on the file server machines.

This chapter examines the structure of the *File Server* process. First, the set of AFS agents with which it must interact are discussed. Next, the threading structure of the server is examined. Some details of its handling of the race conditions created by the callback mechanism are then presented. This is followed by a discussion of the read-only volume synchronization mechanism. This functionality is used in each RPC interface call and intended to detect new releases of read-only volumes. *File Server*s do *not* generate callbacks for objects residing in read-only volumes, so this synchronization information is used to implement a "whole-volume" callback. Finally, the fact that the *File Server* may drop certain information recorded about the *Cache Manager*s with which it has communicated and yet guarantee correctness of operation is explored.

2.2 Interactions

By far the most frequent partner in *File Server* interactions is the set of *Cache Manager*s actively fetching and storing chunks of data files for which the *File Server* provides central storage facilities. The *File Server* also periodically probes the *Cache Manager*s recorded in its tables with which it has recently dealt, determining if they are still active or whether their records might be garbage-collected.

There are two other server entities with which the *File Server* interacts, namely the *Protection Server* and the *BOS Server*. Given a fetch or store request generated by a *Cache Manager*, the *File Server* needs to determine if the caller is authorized to perform the given operation. An important step in this process is to determine what is referred to as the caller's **Current Protection Subdomain**, or **CPS**. A user's CPS is a list of principals, beginning with the user's internal identifier, followed by the the numerical identifiers for all groups to which the user belongs. Once this CPS information is determined, the *File Server* scans the ACL controlling access to the file system object in question. If it finds that the ACL contains an entry specifying a principal with the appropriate rights which also appears in the user's CPS, then the operation is cleared. Otherwise, it is rejected and a protection violation is reported to the *Cache Manager* for ultimate reflection back to the caller.

The *BOS Server* performs administrative operations on the *File Server* process. Thus, their interactions are quite one-sided, and always initiated by the *BOS Server*. The *BOS Server* does not utilize the *File Server*'s RPC interface, but rather generates UNIX signals to achieve the desired effect.

2.3 Threading

The *File Server* is organized as a multi-threaded server. Its threaded behavior within a single UNIX process is achieved by use of the **LWP lightweight process** facility, as described in detail in the companion "AFS-3 Programmer's Reference: Specification for the *Rx* Remote Procedure Call Facility" document. The various threads utilized by the *File Server* are described below:

- **WorkerLWP**: This lightweight process sleeps until a request to execute one of the RPC interface functions arrives. It pulls the relevant information out of the request, including any incoming data delivered as part of the request, and then executes the server stub routine to carry out the operation. The thread finishes its current activation by feeding the return code and any output data back through the RPC channel back to the calling *Cache Manager*. The *File Server* initialization sequence specifies that at least three but no more than six of these `WorkerLWP` threads are to exist at any one time. It is currently not possible to configure the *File Server* process with a different number of `WorkerLWP` threads.

- **FiveMinuteCheckLWP**: This thread runs every five minutes, performing such housekeeping chores as cleaning up timed-out callbacks, setting disk usage statistics, and executing the special handling required by certain AIX implementations. Generally, this thread performs activities that do not take unbounded time to accomplish and do not block the thread. If reassurance is required, `FiveMinuteCheckLWP` can also be told to print out a banner message to the machine's console every so often, stating that the *File Server* process is still running. This is not strictly necessary and an artifact from earlier versions, as the *File Server*'s status is now easily accessible at any time through the *BOS Server* running on its machine.

- **HostCheckLWP**: This thread, also activated every five minutes, performs periodic checking of the status of *Cache Manager*s that have been previously contacted and thus appear in this *File Server*'s internal tables. It generates *RXAFSCB_Probe()* calls from the *Cache Manager* interface, and may find itself suspended for an arbitrary amount of time when it enounters unreachable *Cache Manager*s.

2.4 Callback Race Conditions

Callbacks serve to implement the efficient AFS cache consistency mechanism, as described in Section 1.1.1. Because of the asynchronous nature of callback generation and the multi-threaded operation and organization of both the *File Server* and *Cache Manager*, race conditions can arise in their use. As an example, consider the case of a client machine fetching a chunk of file *X*. The *File Server* thread activated to carry out the operation ships the contents of the chunk and the callback information over to the requesting *Cache Manager*. Before the corresponding *Cache Manager* thread involved in the exchange can be scheduled, another request arrives at the *File Server*, this time storing a modified image of the same chunk from file *X*. Another worker thread comes to life and completes processing of this second request, including execution of an *RXAFSCB_CallBack()* to the *Cache Manager* who still hasn't picked up on the results of its fetch operation. If the *Cache Manager* blindly honors the *RXAFSCB_CallBack()* operation first and then proceeds to process the fetch, it will wind up believing it has a callback on *X* when in reality it is out of sync with the central copy on the *File Server*. To resolve the above class of callback race condition, the *Cache Manager* effectively doublechecks the callback information received from *File Server* calls, making sure they haven't already been nullified by other file system activity.

2.5 Read-Only Volume Synchronization

The *File Server* issues a callback for each file chunk it delivers from a read-write volume, thus allowing *Cache Manager*s to efficiently synchronize their local caches with the authoritative *File Server* images. However, no callbacks are issued when data from read-only volumes is delivered to clients. Thus, it is possible for a new snapshot of the read-only volume to be propagated to the set of replication sites without *Cache Manager*s becoming aware of the event and marking the appropriate chunks in their caches as stale. Although the *Cache Manager* refreshes its volume version information periodically (once an hour), there is still a window where a *Cache Manager* will fail to notice that it has outdated chunks.

The **volume synchronization** mechanism was defined to close this window, resulting in what is nearly a "whole-volume" callback device for read-only volumes. Each *File Server* RPC interface function handling the transfer of file data is equipped with a parameter (*a_volSyncP*), which carries this volume synchronization information. This parameter is set to a non-zero value by the *File Server* exclusively when the data being fetched is coming from a read-only volume. Although the `struct AFSVolSync` defined in Section 5.1.2.2 passed via *a_volSyncP* consists of six longwords, only the first one is set. This leading longword carries the creation date of the read-only volume. The *Cache Manager* immediately compares the synchronization value stored in its cached volume information against the one just received. If they are identical, then the operation is free to complete, secure in the knowledge that all the information and files held from that volume are still current. A mismatch, though, indicates that every file chunk from this volume is potentially out of date, having come from a previous release of the read-only volume. In this case, the *Cache Manager* proceeds to mark every chunk from this volume as suspect. The next time the *Cache Manager* considers accessing any of these chunks, it first checks with the *File Server* it came from which the chunks were obtained to see if they are up to date.

2.6 Disposal of *Cache Manager* Records

Every *File Server*, when first starting up, will, by default, allocate enough space to record 20,000 callback promises (see Section 5.3 for how to override this default). Should the *File Server* fully populate its callback records, it will not allocate more, allowing its memory image to possibly grow in an unbounded fashion. Rather, the *File Server* chooses to break callbacks until it acquires a free record. All reachable *Cache Manager*s respond by marking their cache entries appropriately, preserving the consistency guarantee. In fact, a *File Server* may arbitrarily and unilaterally purge itself of all records associated with a particular *Cache Manager*. Such actions will reduce its performance (forcing these *Cache Manager*s to revalidate items cached from that *File Server*) without sacrificing correctness.

Chapter 3

Cache Manager **Architecture**

3.1 Overview

The AFS *Cache Manager* is a kernel-resident agent with the following duties and responsibilities:

- Users are to be given the illusion that files stored in the AFS distributed file system are in fact part of the local UNIX file system of their client machine. There are several areas in which this illusion is not fully realized:

  - **Semantics**: Full UNIX semantics are not maintained by the set of agents implementing the AFS distributed file system. The largest deviation involves the time when changes made to a file are seen by others who also have the file open. In AFS, modifications made to a cached copy of a file are not necessarily reflected immediately to the central copy (the one hosted by *File Server* disk storage), and thus to other cache sites. Rather, the changes are only guaranteed to be visible to others who simultaneously have their own cached copies open when the modifying process executes a UNIX *close()* operation on the file.

    This differs from the semantics expected from the single-machine, local UNIX environment, where writes performed on one open file descriptor are immediately visible to all processes reading the file via their own file descriptors. Thus, instead of the standard "last writer wins" behavior, users see "last closer wins" behavior on their AFS files. Incidentally, other DFSs, such as NFS, do not implement full UNIX semantics in this case either.

- **Partial failures**: A panic experienced by a local, single-machine UNIX file system will, by definition, cause all local processes to terminate immediately. On the other hand, any hard or soft failure experienced by a *File Server* process or the machine upon which it is executing does not cause any of the *Cache Manager*s interacting with it to crash. Rather, the *Cache Manager*s will now have to reflect their failures in getting responses from the affected *File Server* back up to their callers. Network partitions also induce the same behavior. From the user's point of view, part of the file system tree has become inaccessible. In addition, certain system calls (e.g., *open()* and *read()*) may return unexpected failures to their users. Thus, certain coding practices that have become common amongst experienced (single-machine) UNIX programmers (e.g., not checking error codes from operations that "can't" fail) cause these programs to misbehave in the face of partial failures.

To support this transparent access paradigm, the *Cache Manager* proceeds to:

- Intercept all standard UNIX operations directed towards AFS objects, mapping them to references aimed at the corresponding copies in the local cache.
- Keep a synchronized local cache of AFS files referenced by the client machine's users. If the chunks involved in an operation reading data from an object are either stale or do not exist in the local cache, then they must be fetched from the *File Server*(s) on which they reside. This may require a query to the volume location service in order to locate the place(s) of residence. Authentication challenges from *File Server*s needing to verify the caller's identity are handled by the *Cache Manager*, and the chunk is then incorporated into the cache.
- Upon receipt of a UNIX close, all dirty chunks belonging to the object will be flushed back to the appropriate *File Server*.
- Callback deliveries and withdrawals from *File Server*s must be processed, keeping the local cache in close synchrony with the state of affairs at the central store.

- Interfaces are also be provided for those principals who wish to perform AFS-specific operations, such as Access Control List (ACL) manipulations or changes to the *Cache Manager*'s configuration.

This chapter takes a tour of the *Cache Manager*'s architecture, and examines how it supports these roles and responsibilities. First, the set of AFS agents with which it must interact are discussed. Next, some of the *Cache Manager*'s implementation and interface choices are examined. Finally, the server's ability to arbitrarily dispose of callback information without affecting the correctness of the cache consistency algorithm is explained.

3.2 Interactions

The main AFS agent interacting with a *Cache Manager* is the *File Server*. The most common operation performed by the *Cache Manager* is to act as its users' agent in fetching and storing files to and from the centralized repositories. Related to this activity, a *Cache Manager* must be prepared to answer queries from a *File Server* concerning its health. It must also be able to accept callback revocation notices generated by *File Server*s. Since the *Cache Manager* not only engages in data transfer but must also determine where the data is located in the first place, it also directs inquiries to *Volume Location Server* agents. There must also be an interface allowing direct interactions with both common and administrative users. Certain AFS-specific operations must be made available to these parties. In addition, administrative users may desire to dynamically reconfigure the *Cache Manager*. For example, information about a newly-created cell may be added without restarting the client's machine.

3.3 Implementation Techniques

The above roles and behaviors for the *Cache Manager* influenced the implementation choices and methods used to construct it, along with the desire to maximize portability. This section begins by showing how the VFS/vnode interface, pioneered and standardized by Sun Microsystems, provides not only the necessary fine-grain access to user file system operations, but also facilitates *Cache Manager* ports to new hardware and operating system platforms. Next, the use of UNIX system calls is examined. Finally, the threading structure employed is described.

3.3.1 VFS Interface

As mentioned above, Sun Microsystems has introduced and propagated an important concept in the file system world, that of the Virtual File System (VFS) interface. This abstraction defines a core collection of file system functions which cover all operations required for users to manipulate their data. System calls are written in terms of these standardized routines. Also, the associated *vnode* concept generalizes the original UNIX *inode* idea and provides hooks for differing underlying environments. Thus, to port a system to a new hardware platform, the system programmers have only to construct implementations of this base array of functions consistent with the new underlying machine.

The VFS abstraction also allows multiple file systems (e.g., vanilla UNIX, DOS, NFS, and

AFS) to coexist on the same machine without interference. Thus, to make a machine AFS-capable, a system designer first extends the base vnode structure in well-defined ways in order to store AFS-specific operations with each file description. Then, the base function array is coded so that calls upon the proper AFS agents are made to accomplish each function's standard objectives. In effect, the *Cache Manager* consists of code that interprets the standard set of UNIX operations imported through this interface and executes the AFS protocols to carry them out.

3.3.2 System Calls

As mentioned above, many UNIX system calls are implemented in terms of the base function array of vnode-oriented operations. In addition, one existing system call has been modified and two new system calls have been added to perform AFS-specific operations apart from the *Cache Manager*'s UNIX "emulation" activities. The standard *ioctl()* system call has been augmented to handle AFS-related operations on objects accessed via open UNIX file descriptors. One of the brand-new system calls is **pioctl()**, which is much like *ioctl()* except it names targeted objects by pathname instead of file descriptor. Another is *afs_call()*, which is used to initialize the *Cache Manager* threads, as described in the section immediately following.

3.3.3 Threading

In order to execute its many roles, the *Cache Manager* is organized as a multi-threaded entity. It is implemented with (potentially multiple instantiations of) the following three thread classes:

- **CallBack Listener**: This thread implements the *Cache Manager* callback RPC interface, as described in Section 6.5.
- **Periodic Maintenance**: Certain maintenance and checkup activities need to be performed at five set intervals. Currently, the frequency of each of these operations is hard-wired. It would be a simple matter, though, to make these times configurable by adding command-line parameters to the *Cache Manager*.
  - *Thirty seconds*: Flush pending writes for NFS clients coming in through the NFS-AFS Translator facility.
  - *One minute*: Make sure local cache usage is below the assigned quota, write out dirty buffers holding directory data, and keep *flock()*s alive.

- *Three minutes*: Check for the resuscitation of *File Server*s previously determined to be down, and check the cache of previously computed access information in light of any newly expired tickets.
- *Ten minutes*: Check health of all *File Server*s marked as active, and garbage-collect old RPC connections.
- *One hour*: Check the status of the root AFS volume as well as all cached information concerning read-only volumes.

• **Background Operations**: The *Cache Manager* is capable of prefetching file system objects, as well as carrying out delayed stores, occurring sometime after a *close()* operation. At least two threads are created at *Cache Manager* initialization time and held in reserve to carry out these objectives. This class of background threads implements the following three operations:

- *Prefetch operation*: Fetches particular file system object chunks in the expectation that they will soon be needed.
- *Path-based prefetch operation*: The prefetch daemon mentioned above operates on objects already at least partly resident in the local cache, referenced by their vnode. The path-based prefetch daemon performs the same actions, but on objects named solely by their UNIX pathname.
- *Delayed store operation*: Flush all modified chunks from a file system object to the appropriate *File Server*'s disks.

3.4 Disposal of *Cache Manager* Records

The *Cache Manager* is free to throw away any or all of the callbacks it has received from the set of *File Server*s from which it has cached files. This housecleaning does not in any way compromise the correctness of the AFS cache consistency algorithm. The *File Server* RPC interface described in this paper provides a call to allow a *Cache Manager* to advise of such unilateral jettisoning. However, failure to use this routine still leaves the machine's cache consistent. Let us examine the case of a *Cache Manager* on machine C disposing of its callback on file X from *File Server* F. The next user access on file X on machine C will cause the *Cache Manager* to notice that it does not currently hold a callback on it (although the *File Server* will *think* it does). The *Cache Manager* on C attempts to revalidate its entry when it is entirely possible that the file is still in sync with the central store. In response, the *File Server* will extend the existing callback information it has and deliver the new promise to the *Cache Manager* on C. Now consider the case where file X is modified by a party on a machine other than C before such an access occurs on *C*. Under these circumstances, the *File Server* will break its callback on file *X* before performing the central update. The *Cache Manager* on *C* will receive one of these "break callback" messages. Since it no longer has a callback on file *X*, the *Cache Manager* on *C* will cheerfully acknowledge the *File Server*'s notification and move on to other matters. In either case, the callback information for both parties will eventually resynchronize. The only potential penalty paid is extra inquiries by the *Cache Manager* and thus providing for reduced performance instead of failure of operation.

Chapter 4

Common Definitions and Data Structures

This chapter discusses the definitions used in common by the *File Server* and the *Cache Manager*. They appear in the *common.xg* file, used by *Rxgen* to generate the C code instantiations of these definitions.

4.1 File-Related Definitions

4.1.1 struct AFSFid

This is the type for file system objects within AFS.

Fields

`unsigned long Volume` - This provides the identifier for the volume in which the object resides.

`unsigned long Vnode` - This specifies the index within the given volume corresponding to the object.

`unsigned long Unique` - This is a "uniquifier" or generation number for the slot identified by the `Vnode` field.

4.2 Callback-related Definitions

4.2.1 Types of Callbacks

There are three types of callbacks defined by AFS-3:

- `EXCLUSIVE`: This version of callback has not been implemented. Its intent was to allow a single *Cache Manager* to have exclusive rights on the associated file data.
- `SHARED`: This callback type indicates that the status information kept by a *Cache Manager* for the associated file is up to date. All cached chunks from this file whose version numbers match the status information are thus guaranteed to also be up to date. This type of callback is non-exclusive, allowing any number of other *Cache Manager*s to have callbacks on this file and cache chunks from the file.
- `DROPPED`: This is used to indicate that the given callback promise has been cancelled by the issuing *File Server*. The *Cache Manager* is forced to mark the status of its cache entry as unknown, forcing it to stat the file the next time a user attempts to access any chunk from it.

4.2.2 struct AFSCallBack

This is the canonical callback structure passed in many *File Server* RPC interface calls.

Fields

`unsigned long CallBackVersion` - Callback version number.

`unsigned long ExpirationTime` - Time when the callback expires, measured in seconds.

`unsigned long CallBackType` - The type of callback involved, one of `EXCLUSIVE`, `SHARED`, or `DROPPED`.

4.2.3 Callback Arrays

AFS-3 sometimes does callbacks in bulk. Up to `AFSCBMAX` (50) callbacks can be handled at once. Layouts for the two related structures implementing callback arrays, `struct AFSCBFids` and `struct AFSCBs`, follow below. Note that the callback descriptor in slot i

of the array in the `AFSCBs` structure applies to the file identifier contained in slot i in the fid array in the matching `AFSCBFids` structure.

4.2.3.1 struct AFSCBFids

Fields

`u_int AFSCBFids_len` - Number of AFS file identifiers stored in the structure, up to a maximum of `AFSCBMAX`.

`AFSFid *AFSCBFids_val` - Pointer to the first element of the array of file identifiers.

4.2.3.2 struct AFSCBs

Fields

`u_int AFSCBs_len` - Number of AFS callback descriptors stored in the structure, up to a maximum of `AFSCBMAX`.

`AFSCallBack *AFSCBs_val` - Pointer to the actual array of callback descriptors

4.3 Locking Definitions

4.3.1 struct AFSDBLockDesc

This structure describes the state of an AFS lock.

Fields

`char waitStates` - Types of lockers waiting for the lock.

`char exclLocked` - Does anyone have a boosted, shared or write lock? (A boosted lock allows the holder to have data read-locked and then "boost" up to a write lock on the data without ever relinquishing the lock.)

`char readersReading` - Number of readers that actually hold a read lock on the associated object.

`char numWaiting` - Total number of parties waiting to acquire this lock in some fashion.

4.3.2 struct AFSDBCacheEntry

This structure defines the description of a *Cache Manager* local cache entry, as made accessible via the *RXAFSCB_GetCE()* callback RPC call. Note that *File Server*s do *not* make the above call. Rather, client debugging programs (such as `cmdebug`) are the agents which call *RXAFSCB_GetCE()*.

Fields

`long addr` - Memory location in the *Cache Manager* where this description is located.

`long cell` - Cell part of the fid.

`AFSFid netFid` - Network (standard) part of the fid

`long Length` - Number of bytes in the cache entry.

`long DataVersion` - Data version number for the contents of the cache entry.

`struct AFSDBLockDesc lock` - Status of the lock object controlling access to this cache entry.

`long callback` - Index in callback records for this object.

`long cbExpires` - Time when the callback expires.

`short refCount` - General reference count.

`short opens` - Number of opens performed on this object.

`short writers` - Number of writers active on this object.

`char mvstat` - The file classification, indicating one of normal file, mount point, or volume root.

`char states` - Remembers the state of the given file with a set of bits indicating, from lowest-order to highest order: stat info valid, read-only file, mount point valid, pending core file, wait-for-store, and mapped file.

4.3.3 struct AFSDBLock

This is a fuller description of an AFS lock, including a string name used to identify it.

Fields

`char name[16]` - String name of the lock.

`struct AFSDBLockDesc lock` - Contents of the lock itself.

4.4 Miscellaneous Definitions

4.4.1 Opaque structures

A maximum size for opaque structures passed via the *File Server* interface is defined as `AFSOPAQUEMAX`. Currently, this is set to 1,024 bytes. The `AFSOpaque` typedef is defined for use by those parameters that wish their contents to travel completely uninterpreted across the network.

4.4.2 String Lengths

Two common definitions used to specify basic AFS string lengths are `AFSNAMEMAX` and `AFSPATHMAX`. `AFSNAMEMAX` places an upper limit of 256 characters on such things as file and directory names passed as parameters. `AFSPATHMAX` defines the longest pathname expected by the system, composed of slash-separated instances of the individual directory and file names mentioned above. The longest acceptable pathname is currently set to 1,024 characters.

Chapter 5

File Server **Interfaces**

There are several interfaces offered by the *File Server*, allowing it to export the files stored within the set of AFS volumes resident on its disks to the AFS community in a secure fashion and to perform self-administrative tasks. This chapter will cover the three *File Server* interfaces, summarized below. There is one *File Server* interface that will not be discussed in this document, namely that used by the *Volume Server*. It will be fully described in the companion *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*.

- **RPC**: This is the main *File Server* interface, supporting all of the *Cache Manager*'s needs for providing its own clients with appropriate access to file system objects stored within AFS. It is closedly tied to the callback interface exported by the *Cache Manager* as described in Section 6.5, which has special implications for any application program making direct calls to this interface.

- **Signals**: Certain operations on a *File Server* must be performed by it sending UNIX signals on the machine on which it is executing. These operations include performing clean shutdowns and adjusting debugging output levels. Properly-authenticated administrative users do not have to be physically logged into a *File Server* machine to generate these signals. Rather, they may use the RPC interface exported by that machine's *BOS Server* process to generate them from any AFS-capable machine.

- **Command Line**: Many of the *File Server*'s operating parameters may be set upon startup via its command line interface. Such choices as the number of data buffers and callback records to hold in memory may be made here, along with various other decisions such as lightweight thread stack size.

5.1 RPC Interface

5.1.1 Introduction and Caveats

The documentation for the AFS-3 *File Server* RPC interface commences with some basic definitions and data structures used in conjunction with the function calls. This is followed by an examination of the set of *non-streamed* RPC functions, namely those routines whose parameters are all fixed in size. Next, the *streamed* RPC functions, those with parameters that allow an arbitrary amount of data to be delivered, are described. A code fragment and accompanying description and analysis are offered as an example of how to use the streamed RPC calls. Finally, a description of the special requirements on any application program making direct calls to this *File Server* interface appears. The *File Server* assumes that any entity making calls to its RPC functionality is a bona fide and full-fledged *Cache Manager*. Thus, it expects this caller to export the *Cache Manager*'s own RPC interface, even if the application simply uses *File Server* calls that don't transfer files and thus generate callbacks.

Within those sections describing the RPC functions themselves, the purpose of each call is detailed, and the nature and use of its parameters is documented. Each of these RPC interface routines returns an integer error code, and *a subset* of the possible values are described. A complete and systematic list of potential error returns for each function is difficult to construct and unwieldy to examine. This is due to fact that error codes from many different packages and from many different levels may arise. Instead of attempting completeness, the error return descriptions discuss error codes generated within the functions themselves (or a very small number of code levels below them) within the *File Server* code itself, and not from such associated packages as the *Rx*, volume, and protection modules. Many of these error code are defined in the companion AFS-3 documents.

By convention, a return value of zero reveals that the function call was successful and that all of its `OUT` parameters have been set by the *File Server*.

5.1.2 Definitions and Structures

5.1.2.1 Constants and Typedefs

The following constants and typedefs are required to properly use the *File Server* RPC interface, both to provide values and to interpret information returned by the calls. The constants appear first, followed by the list of typedefs, which sometimes depend on the constants above. Items are alphabetized within each group.

All of the constants appearing below whose names contain the `XSTAT` string are used in conjuction with the *extended data collection facility* supported by the *File Server*. The *File Server* defines some number of *data collections*, each of which consists of an array of longword values computed by the *File Server*.

There are currently three data collections defined for the *File Server*. The first is identified by the `AFS_XSTATSCOLL_CALL_INFO` constant. This collection of longwords relates the number of times each internal function within the *File Server* code has been executed, thus providing profiling information. The second *File Server* data collection is identified by the `AFS_XSTATSCOLL_PERF_INFO` constant. This set of longwords contains information related to the *File Server*'s performance. The third *File Server* data collection is identified by the `AFS_XSTATSCOLL_FULL_PERF_INFO` constant. It contains the extended information related to the *File Server*'s performance.

5.1.2.1.1 AFS_DISKNAMESIZE *[Value = 32]* Specifies the maximum length for an AFS disk partition, used directly in the definition for the `DiskName` typedef. A `DiskName` appears as part of a `struct ViceDisk`, a group of which appear inside a `struct ViceStatistics`, used for carrying basic *File Server* statistics information.

5.1.2.1.2 AFS_MAX_XSTAT_LONGS *[Value = 1,024]* Defines the maximum size for a *File Server* data collection, as exported via the *RXAFS_GetXStats()* RPC call. It is used directly in the `AFS_CollData` typedef.

5.1.2.1.3 AFS_XSTATSCOLL_CALL_INFO *[Value = 0]* This constant identifies the *File Server*'s data collection containing profiling information on the number of times each of its internal procedures has been called.

Please note that this data collection is *not* supported by the *File Server* at this time. A request for this data collection will result the return of a zero-length array.

5.1.2.1.4 AFS_XSTATSCOLL_PERF_INFO *[Value = 1]* This constant identifies the *File Server*'s data collection containing performance-related information.

5.1.2.1.5 AFS_XSTATSCOLL_FULL_PERF_INFO *[Value = 2]* This constant identifies the *File Server*'s data collection containing extended performance-related information.

5.1.2.1.6 AFS_CollData *[typedef long AFS_CollData<AFS_MAX_XSTAT_LONGS>;]* This typedef is used by *Rxgen* to create a structure used to pass *File Server* data collections to the caller. It resolves into a C typedef statement defining a structure of the same name with the following fields:

Fields

- `u_int AFS_CollData_len` - The number of longwords contained within the data pointed to by the next field.
- `long *AFS_CollData_val` - A pointer to a sequence of `AFS_CollData_len` longwords.

5.1.2.1.7 AFSBulkStats *[typedef AFSFetchStatus AFSBulkStats<AFSCBMAX>;]* This typedef is used by *Rxgen* to create a structure used to pass a set of statistics structures, as described in the *RXAFS_BulkStatus* documentation in Section 5.1.3.21. It resolves into a C typedef statement defining a structure of the same name with the following fields:

Fields

- `u_int AFSBulkStats_len` - The number of `struct AFSFetchStatus` units contained within the data to which the next field points.
- `AFSFetchStatus *AFSBulkStats_val` - This field houses pointer to a sequence of `AFSBulkStats_len` units of type `struct AFSFetchStatus`.

5.1.2.1.8 DiskName *[typedef opaque DiskName[AFS_DISKNAMESIZE];]* The name of an AFS disk partition. This object appears as a field within a `struct ViceDisk`, a group of which appear inside a `struct ViceStatistics`, used for carrying basic *File Server* statistics information. The term *opaque* appearing above inidcates that the object being defined will be treated as an undifferentiated string of bytes.

5.1.2.1.9 ViceLockType *[typedef long ViceLockType;]* This defines the format of a lock used internally by the *Cache Manager*. The content of these locks is accessible via the *RXAFSCB_GetLock()* RPC function. An isomorphic and more refined version of the lock structure used by the *Cache Manager*, mapping directly to this definition, is `struct AFSDBLockDesc`, defined in Section 4.3.1.

5.1.2.2 struct AFSVolSync

This structure conveys volume synchronization information across many of the *File Server* RPC interface calls, allowing something akin to a 'whole-volume callback" on read-only volumes.

Fields

`unsigned long spare1 ... spare6` - The first longword, `spare1`, contains the volume's creation date. The rest are currently unused.

5.1.2.3 struct AFSFetchStatus

This structure defines the information returned when a file system object is fetched from a *File Server*.

Fields

`unsigned long InterfaceVersion` - RPC interface version, defined to be 1.

`unsigned long FileType` - Distinguishes the object as either a file, directory, symlink, or invalid.

`unsigned long LinkCount` - Number of links to this object.

`unsigned long Length` - Length in bytes.

`unsigned long DataVersion` - Object's data version number.

`unsigned long Author` - Identity of the object's author.

`unsigned long Owner` - Identity of the object's owner.

`unsigned long CallerAccess` - The set of access rights computed for the caller on this object.

`unsigned long AnonymousAccess` - The set of access rights computed for any completely unauthenticated principal.

`unsigned long UnixModeBits` - Contents of associated UNIX mode bits.

`unsigned long ParentVnode` - Vnode for the object's parent directory.

`unsigned long ParentUnique` - Uniquifier field for the parent object.

`unsigned long SegSize` - *(Not implemented).*

`unsigned long ClientModTime` - Time when the caller last modified the data within the object.

`unsigned long ServerModTime` - Time when the server last modified the data within the object.

`unsigned long Group` - *(Not implemented).*

`unsigned long SyncCounter` - *(Not implemented).*

`unsigned long spare1 ... spare4` - Spares.

5.1.2.4 struct AFSStoreStatus

This structure is used to convey which of a file system object's status fields should be set, and their new values. Several *File Server* RPC calls, including *RXAFS_StoreStatus()*, *RXAFS_CreateFile()*, *RXAFS_SymLink()*, *RXAFS_MakeDir()*, and the streamed call to store file data onto the *File Server*.

Fields

`unsigned long Mask` - Bit mask, specifying which of the following fields should be assigned into the *File Server*'s status block on the object.

`unsigned long ClientModTime` - The time of day that the object was last modified.

`unsigned long Owner` - The principal identified as the owner of the file system object.

`unsigned long Group` - *(Not implemented).*

`unsigned long UnixModeBits` - The set of associated `UNIX` mode bits.

`unsigned long SegSize` - *(Not implemented).*

5.1.2.5 struct ViceDisk

This structure occurs in `struct ViceStatistics`, and describes the characteristics and status of a disk partition used for AFS storage.

Fields

`long BlocksAvailable` - Number of 1 Kbyte disk blocks still available on the partition.

`long TotalBlocks` - Total number of disk blocks in the partition.

`DiskName Name` - The human-readable character string name of the disk partition (e.g., `/vicepa`).

5.1.2.6 struct ViceStatistics

This is the *File Server* statistics structure returned by the *RXAFS_GetStatistics()* RPC call.

Fields

`unsigned long CurrentMsgNumber` - *Not used*

`unsigned long OldestMsgNumber` - *Not used*

`unsigned long CurrentTime` - Time of day, as understood by the *File Server*.

`unsigned long BootTime` - Kernel's boot time.

`unsigned long StartTime` - Time when the *File Server* started up.

`long CurrentConnections` - Number of connections to *Cache Manager* instances.

`unsigned long TotalViceCalls` - Count of all calls made to the RPC interface.

`unsigned long TotalFetchs` - Total number of fetch operations, either status or data, performed.

`unsigned long FetchDatas` - Total number of data fetch operations exclusively.

`unsigned long FetchedBytes` - Total number of bytes fetched from the *File Server* since it started up.

`long FetchDataRate` - Result of dividing the `FetchedBytes` field by the number of seconds the *File Server* has been running.

`unsigned long TotalStores` - Total number of store operations, either status or data, performed.

`unsigned long StoreDatas` - Total number of data store operations exclusively.

`unsigned long StoredBytes` - Total number of bytes stored to the *File Server* since it started up.

`long StoreDataRate` - The result of dividing the `StoredBytes` field by the number of seconds the *File Server* has been running.

`unsigned long TotalRPCBytesSent` - *Outdated*

`unsigned long TotalRPCBytesReceived` - *Outdated*

`unsigned long TotalRPCPacketsSent` - *Outdated*

`unsigned long TotalRPCPacketsReceived` - *Outdated*

`unsigned long TotalRPCPacketsLost` - *Outdated*

`unsigned long TotalRPCBogusPackets` - *Outdated*

`long SystemCPU` - Result of reading from the kernel the usage times attributed to system activities.

`long UserCPU` - Result of reading from the kernel the usage times attributed to user-level activities.

`long NiceCPU` - Result of reading from the kernel the usage times attributed to *File Server* activities that have been *nice()*d (i.e., run at a lower priority).

`long IdleCPU` - Result of reading from the kernel the usage times attributed to idling activities.

`long TotalIO` - Summary of the number of bytes read/written from the disk.

`long ActiveVM` - Amount of virtual memory used by the *File Server*.

`long TotalVM` - Total space available on disk for virtual memory activities.

`long EtherNetTotalErrors` - *Not used.*

`long EtherNetTotalWrites` - *Not used.*

`long EtherNetTotalInterupts` - *Not used.*

`long EtherNetGoodReads` - *Not used.*

`long EtherNetTotalBytesWritten` - *Not used.*

`long EtherNetTotalBytesRead` - *Not used.*

`long ProcessSize` - The size of the *File Server*'s data space in 1 Kbyte chunks.

`long WorkStations` - The total number of client *Cache Manager*s (workstations) for which information is held by the *File Server*.

`long ActiveWorkStations` - The total number of client *Cache Manager*s (workstations) that have recently interacted with the *File Server*. This number is strictly less than or equal to the `WorkStations` field.

`long Spare1 ... Spare8` - *Not used.*

`ViceDisk Disk1 ... Disk10` - Statistics concerning up to 10 disk partitions used by the *File Server*. These records keep information on *all* partitions, not just partitions reserved for AFS storage.

5.1.2.7 struct afs_PerfStats

This is the structure corresponding to the `AFS_XSTATSCOLL_PERF_INFO` data collection that is defined by the *File Server* (see Section 5.1.2.1.4). It is accessible via the *RXAFS_GetXStats()* interface routine, as defined in Section 5.1.3.26.

The fields within this structure fall into the following classifications:

- Number of requests for the structure.
- Vnode cache information.
- Directory package numbers.
- Rx information.
- Host module fields.
- Unique hardware/OS identifier.
- Spares.

Please note that the *Rx* fields represent the contents of the `rx_stats` structure maintained by the *Rx* RPC facility itself. Also, a full description of all the structure's fields is not possible here. For example, the reader is referred to the companion *Rx* document for further clarification on the *Rx*-related fields within `afs_PerfStats`.

Fields

`long numPerfCalls` - Number of performance collection calls received.

`long vcache_L_Entries` - Number of entries in large (directory) vnode cache.

`long vcache_L_Allocs` - Number of allocations for the large vnode cache.

`long vcache_L_Gets` - Number of get operations for the large vnode cache.

`long vcache_L_Reads` - Number of reads performed on the large vnode cache.

`long vcache_L_Writes` - Number of writes executed on the large vnode cache.

`long vcache_S_Entries` - Number of entries in the small (file) vnode cache.

`long vcache_S_Allocs` - Number of allocations for the small vnode cache.

`long vcache_S_Gets` - Number of get operations for the small vnode cache.

`long vcache_S_Reads` - Number of reads performed on the small vnode cache.

`long vcache_S_Writes` - Number of writes executed on the small vnode cache.

`long vcache_H_Entries` - Number of entries in the header of the vnode cache.

`long vcache_H_Gets` - Number of get operations on the header of the vnode cache.

`long vcache_H_Replacements` - Number of replacement operations on the header of the vnode cache.

`long dir_Buffers` - Number of directory package buffers in use.

`long dir_Calls` - Number of read calls made to the directory package.

`long dir_IOs` - Number of directory I/O operations performed.

`long rx_packetRequests` - Number of *Rx* packet allocation requests.

`long rx_noPackets_RcvClass` - Number of failed packet reception requests.

`long rx_noPackets_SendClass` - Number of failed packet transmission requests.

`long rx_noPackets_SpecialClass` - Number of “special” *Rx* packet requests.

`long rx_socketGreedy` - Did setting the *Rx* socket to `SO_GREEDY` succeed?

`long rx_bogusPacketOnRead` - Number of short packets received.

`long rx_bogusHost` - Latest host address from bogus packets.

`long rx_noPacketOnRead` - Number of attempts to read a packet when one was not physically available.

`long rx_noPacketBuffersOnRead` - Number of packets dropped due to buffer shortages.

`long rx_selects` - Number of selects performed waiting for a packet arrival or a timeout.

`long rx_sendSelects` - Number of selects forced upon a send.

`long rx_packetsRead_RcvClass` - Number of packets read belonging to the “Rcv” class.

`long rx_packetsRead_SendClass` - Number of packets read that belong to the “Send” class.

`long rx_packetsRead_SpecialClass` - Number of packets read belonging to the “Special” class.

`long rx_dataPacketsRead` - Number of unique data packets read off the wire.

`long rx_ackPacketsRead` - Number of acknowledgment packets read.

`long rx_dupPacketsRead` - Number of duplicate data packets read.

`long rx_spuriousPacketsRead` - Number of inappropriate packets read.

`long rx_packetsSent_RcvClass` - Number of packets sent belonging to the "Rcv" class.

`long rx_packetsSent_SendClass` - Number of packets sent belonging to the "Send" class.

`long rx_packetsSent_SpecialClass` - Number of packets sent belonging to the "Special" class.

`long rx_ackPacketsSent` - Number of acknowledgment packets sent.

`long rx_pingPacketsSent` - Number of ping packets sent.

`long rx_abortPacketsSent` - Number of abort packets sent.

`long rx_busyPacketsSent` - Number of busy packets sent.

`long rx_dataPacketsSent` - Number of unique data packets sent.

`long rx_dataPacketsReSent` - Number of retransmissions sent.

`long rx_dataPacketsPushed` - Number of retransmissions pushed by a `NACK`.

`long rx_ignoreAckedPacket` - Number of packets whose `acked` flag was set at *rxi_Start()* time.

`long rx_totalRtt_Sec` - Total round trip time in seconds.

`long rx_totalRtt_Usec` - Microsecond portion of the total round trip time.

`long rx_minRtt_Sec` - Minimum round trip time in seconds.

`long rx_minRtt_Usec` - Microsecond portion of minimal round trip time.

`long rx_maxRtt_Sec` - Maximum round trip time in seconds.

`long rx_maxRtt_Usec` - Microsecond portion of maximum round trip time.

`long rx_nRttSamples` - Number of round trip samples.

`long rx_nServerConns` - Total number of server connections.

`long rx_nClientConns` - Total number of client connections.

`long rx_nPeerStructs` - Total number of peer structures.

`long rx_nCallStructs` - Total number of call structures.

`long rx_nFreeCallStructs` - Total number of call structures residing on the free list.

`long host_NumHostEntries` - Number of host entries.

`long host_HostBlocks` - Number of blocks in use for host entries.

`long host_NonDeletedHosts` - Number of non-deleted host entries.

`long host_HostsInSameNetOrSubnet` - Number of host entries in the same subnet as the *File Server*.

`long host_HostsInDiffSubnet` - Number of host entries in a different subnet as the *File Server*.

`long host_HostsInDiffNetwork` - Number of host entries in an entirely different network than the *File Server*.

`long host_NumClients` - Number of client entries.

`long host_ClientBlocks` - Number of blocks in use for client entries.

`long sysname_ID` - Unique hardware/OS identifier.

`long spare[31]` - Spare fields, reserved for future use.

5.1.2.8 stuct fs_stats_opTimingData

This is the structure that maintains detailed statistics regarding *File Server* RPC calls.

Fields

`long numOps` - Number of operations executed.

`long numSuccesses` - Number of successful operations.

`struct timeval sumTime` - Sum of sample timings.

`struct timeval sqrTime` - Sum of squares of sample timings.

`struct timeval minTime` - Minimum timing value observed.

`struct timeval maxTime` - Maximum timing value observed.

5.1.2.9 struct fs_stats_xferData

This is the structure that maintains detailed statistics regarding *File Server* data transfer operations.

Fields

`long numXfers` - Number of transfers.

`long numSuccesses` - Number of successful transfers.

`struct timeval sumTime` - Sum of sample timings.

`struct timeval sqrTime` - Sum of squares of sample timings.

`struct timeval minTime` - Minimum sample time recorded.

`struct timeval maxTime` - Maximum sample time recorded.

`long sumBytes` - Sum of bytes transferred (in KB).

`long minBytes` - Minimum value observed.

`long maxBytes` - Maximum value observed.

`long count[FS_STATS_NUM_XFER_BUCKETS]` - Tally for each range of bytes.

The value of `FS_STATS_NUM_XFER_BUCKETS` constant is 9. The array `count` tallies each transfer according to the number of bytes transferred: 0-128B, 129-1024B, etc. The following constants identify the ranges:

```
#define FS_STATS_MAXBYTES_BUCKET0     128
#define FS_STATS_MAXBYTES_BUCKET1    1024
#define FS_STATS_MAXBYTES_BUCKET2    8192
#define FS_STATS_MAXBYTES_BUCKET3   16384
#define FS_STATS_MAXBYTES_BUCKET4   32768
#define FS_STATS_MAXBYTES_BUCKET5  131072
#define FS_STATS_MAXBYTES_BUCKET6  524288
#define FS_STATS_MAXBYTES_BUCKET7 1048576
```

For example `count[0]` records number of transfers with 0-128B of data, `count[1]` records number of transfers with 129-1024B of data, and `count[9]` records number of transfers with more than 1,048,576B of data.

Note: Both the `fs_stats_opTimingData` and `fs_stats_xferData` structures contain a field `sqrTime`. This field represents a sum of squares of timings (the simple sum is kept in the `sumTime`) and can be used for computing standard deviation of timings.

5.1.2.10 fs_stats_DetailedStats

This is the structure containing detailed statistics on *File Server* RPC calls and data transfer operations.

File Server Interfaces

Fields

`struct timeval epoch` - Time when data collection began.

`struct fs_stats_opTimingData rpcOpTimes[FS_STATS_NUM_RPC_OPS]` - Individual RPC operation timings.

`struct fs_stats_xferData xferOpTimes[FS_STATS_NUM_XFER_OPS]` - Byte transfer information for data fetch and store operations.

The value of the `FS_STATS_NUM_RPC_OPS` constant is 28 and indices of the `rpcOpTimes` array correspond to the following constants and related RPC calls:

```
#define FS_STATS_RPCIDX_FETCHDATA        0
#define FS_STATS_RPCIDX_FETCHACL         1
#define FS_STATS_RPCIDX_FETCHSTATUS      2
#define FS_STATS_RPCIDX_STOREDATA        3
#define FS_STATS_RPCIDX_STOREACL         4
#define FS_STATS_RPCIDX_STORESTATUS      5
#define FS_STATS_RPCIDX_REMOVEFILE       6
#define FS_STATS_RPCIDX_CREATEFILE       7
#define FS_STATS_RPCIDX_RENAME           8
#define FS_STATS_RPCIDX_SYMLINK          9
#define FS_STATS_RPCIDX_LINK            10
#define FS_STATS_RPCIDX_MAKEDIR         11
#define FS_STATS_RPCIDX_REMOVEDIR       12
#define FS_STATS_RPCIDX_SETLOCK         13
#define FS_STATS_RPCIDX_EXTENDLOCK      14
#define FS_STATS_RPCIDX_RELEASELOCK     15
#define FS_STATS_RPCIDX_GETSTATISTICS   16
#define FS_STATS_RPCIDX_GIVEUPCALLBACKS 17
#define FS_STATS_RPCIDX_GETVOLUMEINFO   18
#define FS_STATS_RPCIDX_GETVOLUMESTATUS 19
#define FS_STATS_RPCIDX_SETVOLUMESTATUS 20
#define FS_STATS_RPCIDX_GETROOTVOLUME   21
#define FS_STATS_RPCIDX_CHECKTOKEN      22
#define FS_STATS_RPCIDX_GETTIME         23
#define FS_STATS_RPCIDX_NGETVOLUMEINFO  24
#define FS_STATS_RPCIDX_BULKSTATUS      25
#define FS_STATS_RPCIDX_XSTATSVERSION   26
#define FS_STATS_RPCIDX_GETXSTATS       27
```

The value of the `FS_STATS_NUM_XFER_OPS` constant is 2 and indices of the `xferOpTimes` array correspond to the following constants:

```
#define FS_STATS_XFERIDX_FETCHDATA       0
#define FS_STATS_XFERIDX_STOREDATA       1
```

5.1.2.11 struct fs_FullPerfStats

This is the structure corresponding to the `AFS_XSTATSCOLL_FULL_PERF_INFO` data collection that is defined by the *File Server* (see Section 5.1.2.1.5). It is accessible via the *RXAFS_GetXStats()* interface routine, as defined in Section 5.1.3.26.

Fields

`struct afs_PerfStats overall` - Overall statistics.

`struct fs_stats_DetailedStats det` - Detailed statistics.

5.1.2.12 struct AFSFetchVolumeStatus

The results of asking the *File Server* for status information concerning a particular volume it hosts.

Fields

`long Vid` - Volume ID.

`long ParentId` - Volume ID in which the given volume is "primarily" mounted. This is used to properly resolve `pwd` operations, as a volume may be mounted simultaneously at multiple locations.

`char Online` - Is the volume currently online and fully available?

`char InService` - This field records whether the volume is currently in service. It is indistinguishable from the `Blessed` field,

`char Blessed` - See the description of the `InService` field immediately above.

`char NeedsSalvage` - Should this volume be salvaged (run through a consistency-checking procedure)?

`long Type` - The classification of this volume, namely a read/write volume (`RWVOL` = 0), read-only volume (`ROVOL` = 1), or backup volume (`BACKVOL` = 2).

`long MinQuota` - Minimum number of 1 Kbyte disk blocks to be set aside for this volume. Note: this field is not currently set or accessed by any AFS agents.

`long MaxQuota` - Maximum number of 1 Kbyte disk blocks that may be occupied by this volume.

`long BlocksInUse` - Number of 1 Kbyte disk blocks currently in use by this volume.

`long PartBlocksAvail` - Number of available 1 Kbyte blocks currently unused in the volume's partition.

`long PartMaxBlocks` - Total number of blocks, in use or not, for the volume's partition.

5.1.2.13 struct AFSStoreVolumeStatus

This structure is used to convey which of a file system object's status fields should be set, and their new values. The *RXAFS_SetVolumeStatus()* RPC call is the only user of this structure.

Fields

`long Mask` - Bit mask to determine which of the following two fields should be stored in the centralized status for a given volume.

`long MinQuota` - Minimum number of 1 Kbyte disk blocks to be set aside for this volume.

`long MaxQuota` - Maximum number of 1 Kbyte disk blocks that may be occupied by this volume.

5.1.2.14 struct AFSVolumeInfo

This field conveys information regarding a particular volume through certain *File Server* RPC interface calls. For information regarding the different volume types that exist, please consult the companion document, AFS-3 Programmer's Reference:*Volume Server/Volume Location Server* Interface.

Fields

`unsigned long Vid` - Volume ID.

`long Type` - Volume type (see `struct AFSFetchVolumeStatus` in Section 5.1.2.12 above).

`unsigned long Type0 ... Type4` - The volume IDs for the possible volume types in existance for this volume.

`unsigned long ServerCount` - The number of *File Server* machines on which an instance of this volume is located.

`unsigned long Server0 ... Server7` - Up to 8 IP addresses of *File Server* machines hosting an instance on this volume. The first `ServerCount` of these fields hold valid server addresses.

`unsigned short Port0 ... Port7` - Up to 8 UDP port numbers on which operations on this volume should be directed. The first `ServerCount` of these fileds hold valid port identifiers.

5.1.3 Non-Streamed Function Calls

The following is a description of the *File Server* RPC interface routines that utilize only parameters with fixed maximum lengths. The majority of the *File Server* calls fall into this suite, with only a handful using streaming techniques to pass objects of unbounded size between a *File Server* and *Cache Manager*.

Each function is labeled with an *opcode* number. This is the low-level numerical identifier for the function, and appears in the set of network packets constructed for the RPC call.

5.1.3.1 RXAFS_FetchACL — Fetch the ACL associated with the given AFS file identifier

```
int RXAFS_FetchACL(IN struct rx_connection *a_rxConnP,
                   IN AFSFid *a_dirFidP,
                   OUT AFSOpaque *a_ACLP,
                   OUT AFSFetchStatus *a_dirNewStatP,
                   OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 131] Fetch the ACL for the directory identified by *a_dirFidP*, placing it in the space described by the opaque structure to which *a_ACLP* points. Also returned is the given directory's status, written to *a_dirNewStatP*. An ACL may thus take up at most `AFSOPAQUEMAX` (1,024) bytes, since this is the maximum size of an `AFSOpaque`.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid was provided.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.2 RXAFS_FetchStatus — Fetch the status information regarding a given file system object

```
int RXAFS_FetchStatus(IN struct rx_connection *a_rxConnP,
                      IN AFSFid *a_fidToStatP,
                      OUT AFSFetchStatus *a_currStatP,
                      OUT AFSCallBack *a_callBackP,
                      OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 132] Fetch the current status information for the file or directory identified by *a_fidToStatP*, placing it into the area to which *a_currStatP* points. If the object resides in a read/write volume, then the related callback information is returned in *a_callBackP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid was provided.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.3 RXAFS_StoreACL — Associate the given ACL with the named directory

```
int RXAFS_StoreACL(IN struct rx_connection *a_rxConnP,
                   IN AFSOpaque *a_ACLToStoreP,
                   IN AFSFid *a_dirFidP,
                   OUT AFSFetchStatus *a_dirNewStatP,
                   OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 134] Store the ACL information to which *a_ACLToStoreP* points to the *File Server*, associating it with the directory identified by *a_dirFidP*. The resulting status information for the *a_dirFidP* directory is returned in *a_dirNewStatP*. Note that the ACL supplied via *a_ACLToStoreP* may be at most `AFSOPAQUEMAX` (1,024) bytes long, since this is the maximum size accommodated by an `AFSOpaque`.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

- `EACCES` The caller is not permitted to perform this operation.
- `E2BIG` The given ACL is too large.
- `EINVAL` The given ACL could not translated to its on-disk format.

5.1.3.4 RXAFS_StoreStatus — Store the given status information for the specified file

```
int RXAFS_StoreStatus(IN struct rx_connection *a_rxConnP,
                      IN AFSFid *a_fidP,
                      IN AFSStoreStatus *a_currStatusP,
                      OUT AFSFetchStatus *a_srvStatusP,
                      OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 135] Store the status information to which *a_currStatusP* points, associating it with the file identified by *a_fidP*. All outstanding callbacks on this object are broken. The resulting status structure stored at the *File Server* is returned in *a_srvStatusP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid was provided, or an attempt was made to change the mode of a symbolic link.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.5 RXAFS_RemoveFile — Delete the given file

```
int RXAFS_RemoveFile(IN struct rx_connection *a_rxConnP,
                     IN AFSFid *a_dirFidP,
                     IN char *a_name<AFSNAMEMAX>,
                     OUT AFSFetchStatus *a_srvStatusP,
                     OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 136] Destroy the file named `a_name` within the directory identified by `a_dirFidP`. All outstanding callbacks on this object are broken. The resulting status structure stored at the *File Server* is returned in *a_srvStatusP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid was provided, or an attempt was made to remove "." or "..".

`EISDIR` The target of the deletion was supposed to be a file, but it is really a directory.

`ENOENT` The named file was not found.

`ENOTDIR` The `a_dirFidP` parameter references an object which is not a directory, or the deletion target is supposed to be a directory but is not.

`ENOTEMPTY` The target directory being deleted is not empty.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.6 RXAFS_CreateFile — Create the given file

```
int RXAFS_CreateFile(IN struct rx_connection *a_rxConnP,
                     IN AFSFid *DirFid,
                     IN char *Name,
                     IN AFSStoreStatus *InStatus,
                     OUT AFSFid *OutFid,
                     OUT AFSFetchStatus *OutFidStatus,
                     OUT AFSFetchStatus *OutDirStatus,
                     OUT AFSCallBack *CallBack,
                     OUT AFSVolSync *a_volSyncP)
```

associated with the new file.

Description

[Opcode 137] This call is used to create a file, but *not* for creating a directory or a symbolic link. If this call succeeds, it is the *Cache Manager*'s responsibility to either create an entry locally in the directory specified by *DirFid* or to invalidate this directory's cache entry.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller is not permitted to perform this operation.

`EINVAL` An internal error in looking up the client record was encountered, or an invalid fid or name was provided.

`ENOTDIR` The `DirFid` parameter references an object which is not a directory.

`VICETOKENDEAD` Caller's authentication token has expired.

5.1.3.7 RXAFS_Rename — Rename the specified file in the given directory

```
int RXAFS_Rename(IN struct rx_connection *a_rxConnP,
                 IN AFSFid *a_origDirFidP,
                 IN char *a_origNameP,
                 IN AFSFid *a_newDirFidP,
                 IN char *a_newNameP,
                 OUT AFSFetchStatus *a_origDirStatusP,
                 OUT AFSFetchStatus *a_newDirStatusP,
                 OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 138] Rename file *a_origNameP* in the directory identified by *a_origDirFidP*. Its new name is to be *a_newNameP*, and it will reside in the directory identified by *a_newDirFidP*. Each of these names must be no more than `AFSNAMEMAX` (256) characters long. The status of the original and new directories after the rename operation completes are deposited in *a_origDirStatusP* and *a_newDirStatusP* respectively. Existing callbacks are broken for all files and directories involved in the operation.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` New file exists but user doesn't have `Delete` rights in the directory.

`EINVAL` Name provided is invalid.

`EISDIR` Original object is a file and new object is a directory.

`ENOENT` The object to be renamed doesn't exist in the parent directory.

`ENOTDIR` Original object is a directory and new object is a file.

`EXDEV` Rename attempted across a volume boundary, or create a pathname loop, or hard links exist to the file.

5.1.3.8 RXAFS_Symlink — Create a symbolic link

```
int RXAFS_Symlink(IN struct rx_connection *a_rxConnP,
                  IN AFSFid *a_dirFidP,
                  IN char *a_nameP,
                  IN char *a_linkContentsP,
                  IN AFSStoreStatus *a_origDirStatP,
                  OUT AFSFid *a_newFidP,
                  OUT AFSFetchStatus *a_newFidStatP,
                  OUT AFSFetchStatus *a_newDirStatP,
                  OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 139] Create a symbolic link named *a_nameP* in the directory identified by *a_dirFidP*. The text of the symbolic link is provided in *a_linkContentsP*, and the desired status fields for the symbolic link given by *a_origDirStatP*. The name offered in *a_nameP* must be less than `AFSNAMEMAX` (256) characters long, and the text of the link to which *a_linkContentsP* points must be less than `AFSPATHMAX` (1,024) characters long. Once the symbolic link has been successfully created, its file identifier is returned in *a_newFidP*. Existing callbacks to the *a_dirFidP* directory are broken before the symbolic link creation completes. The status fields for the symbolic link itself and its parent's directory are returned in *a_newFidStatP* and *a_newDirStatP* respectively.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` Illegal symbolic link name provided.

5.1.3.9 RXAFS_Link — Create a hard link

```
int RXAFS_Link(IN struct rx_connection *a_rxConnP,
               IN AFSFid *a_dirFidP,
               IN char *a_nameP,
               IN AFSFid *a_existingFidP,
               OUT AFSFetchStatus *a_newFidStatP,
               OUT AFSFetchStatus *a_newDirStatP,
               OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 140] Create a hard link named *a_nameP* in the directory identified by *a_dirFidP*. The file serving as the basis for the hard link is identified by *existingFidP*. The name offered in *a_nameP* must be less than `AFSNAMEMAX` (256) characters long. Existing callbacks to the *a_dirFidP* directory are broken before the hard link creation completes. The status fields for the file itself and its parent's directory are returned in *a_newFidStatP* and *a_newDirStatP* respectively.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EISDIR` An attempt was made to create a hard link to a directory.

`EXDEV` Hard link attempted across directories.

5.1.3.10 RXAFS_MakeDir — Create a directory

```
int RXAFS_MakeDir(IN struct rx_connection *a_rxConnP,
                  IN AFSFid *a_parentDirFid,P
                  IN char *a_newDirNameP,
                  IN AFSStoreStatus *a_currStatP,
                  OUT AFSFid *a_newDirFidP,
                  OUT AFSFetchStatus *a_dirFidStatP,
                  OUT AFSFetchStatus *a_parentDirStatP,
                  OUT AFSCallBack *a_newDirCallBackP,
                  OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 141] Create a directory named *a_newDirNameP* within the directory identified by *a_parentDirFidP*. The initial status fields for the new directory are provided in *a_currStatP*. The new directory's name must be less than `AFSNAMEMAX` (256) characters long. The new directory's ACL is inherited from its parent. Existing callbacks on the parent directory are broken before the creation completes. Upon successful directory creation, the new directory's file identifier is returned in *a_newDirFidP*, and the resulting status information for the new and parent directories are stored in *a_dirFidStatP* and *a_parentDirStatP* respectively. In addition, a callback for the new directory is returned in *a_newDirCallBackP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` The directory name provided is unacceptable.

5.1.3.11 RXAFS_RemoveDir — Remove a directory

```
int RXAFS_RemoveDir(IN struct rx_connection *a_rxConnP,
                    IN AFSFid *a_parentDirFidP,
                    IN char *a_dirNameP,
                    OUT AFSFetchStatus *a_newParentDirStatP,
                    OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 142] Remove the directory named *a_dirNameP* from within its parent directory, identified by *a_parentDirFid*. The directory being removed must be empty, and its name must be less than `AFSNAMEMAX` (256) characters long. Existing callbacks to the directory being removed and its parent directory are broken before the deletion completes. Upon successful deletion, the status fields for the parent directory are returned in *a_newParentDirStatP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

5.1.3.12 RXAFS_GetStatistics — Get common *File Server* statistics

```
int RXAFS_GetStatistics(IN struct rx_connection *a_rxConnP,
                        OUT ViceStatistics *a_FSInfoP)
```

Description

[Opcode 146] Fetch the structure containing a set of common *File Server* statistics. These numbers represent accumulated readings since the time the *File Server* last restarted. For a full description of the individual fields contained in this structure, please see Section 5.1.2.6.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`---` No error codes generated.

5.1.3.13 RXAFS_GiveUpCallBacks — Ask the *File Server* to break the given set of callbacks on the corresponding set of file identifiers

```
int RXAFS_GiveUpCallBacks(IN struct rx_connection *a_rxConnP,
                          IN AFSCBFids *a_fidArrayP,
                          IN AFSCBs *a_callBackArrayP)
```

Description

[Opcode 147] Given an array of up to `AFSCBMAX` file identifiers in *a_fidArrayP* and a corresponding number of callback structures in *a_callBackArrayP*, ask the *File Server* to remove these callbacks from its register. Note that this routine only affects callbacks outstanding on the given set of files for the host issuing the *RXAFS_GiveUpCallBacks* call. Callback promises made to other machines on any or all of these files are not affected.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`EINVAL` More file identifiers were provided in the *a_fidArrayP* than callbacks in the *a_callBackArray*.

5.1.3.14 RXAFS_GetVolumeInfo — Get information about a volume given its name

```
int RXAFS_GetVolumeInfo(IN struct rx_connection *a_rxConnP,
                        IN char *a_volNameP,
                        OUT VolumeInfo *a_volInfoP)
```

Description

[Opcode 148] Ask the given *File Server* for information regarding a volume whose name is *a_volNameP*. The volume name must be less than `AFSNAMEMAX` characters long, and the volume itself must reside on the *File Server* being probed.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Please note that definitions for the error codes with `VL_` prefixes may be found in the *vlserver.h* include file

Error Codes

`1` Could not contact any of the corresponding *Volume Location Server*s.

`VL_BADNAME` An improperly-formatted volume name provided.

`VL_ENTDELETED` An entry was found for the volume, reporting that the volume has been deleted.

`VL_NOENT` The given volume was not found.

5.1.3.15 RXAFS_GetVolumeStatus — Get basic status information for the named volume

```
int RXAFS_GetVolumeStatus(IN struct rx_connection *a_rxConnP,
                          IN long a_volIDP,
                          OUT AFSFetchVolumeStatus *a_volFetchStatP,
                          OUT char *a_volNameP,
                          OUT char *a_offLineMsgP,
                          OUT char *a_motdP)
```

Description

[Opcode 149] Given the numeric volume identifier contained in *a_volIDP*, fetch the basic status information corresponding to that volume. This status information is stored into *a_volFetchStatP*. A full description of this status structure is found in Section 5.1.2.12. In addition, three other facts about the volume are returned. The volume's character string name is placed into *a_volNameP*. This name is guaranteed to be less than `AFSNAMEMAX` characters long. The volume's offline message, namely the string recording why the volume is off-line (if it is), is stored in *a_offLineMsgP* . Finally, the volume's "Message of the Day" is placed in *a_motdP*. Each of the character strings deposited into *a_offLineMsgP* and *a_motdP* is guaranteed to be less than `AFSOPAQUEMAX` (1,024) characters long.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` A volume identifier of zero was specified.

5.1.3.16 RXAFS_SetVolumeStatus — Set the basic status information for the named volume

```
int RXAFS_SetVolumeStatus(IN struct rx_connection *a_rxConnP,
                          IN long a_volIDP,
                          IN AFSStoreVolumeStatus *a_volStoreStatP,
                          IN char *a_volNameP,
                          IN char *a_offLineMsgP,
                          IN char *a_motdP)
```

Description

[Opcode 150] Given the numeric volume identifier contained in *a_volIDP*, set that volume's basic status information to the values contained in *a_volStoreStatP*. A full description of the fields settable by this call, including the necessary masking, is found in Section 5.1.2.13. In addition, three other items relating to the volume may be set. Non-null character strings found in *a_volNameP*, *a_offLineMsgP*, and *a_motdP* will be stored in the volume's printable name, off-line message, and "Message of the Day" fields respectively. The volume name provided must be less than `AFSNAMEMAX` (256) characters long, and the other two strings must be less than `AFSOPAQUEMAX` (1,024) characters long each.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` A volume identifier of zero was specified.

5.1.3.17 RXAFS_GetRootVolume — Return the name of the root volume for the file system

```
int RXAFS_GetRootVolume(IN struct rx_connection *a_rxConnP,
                        OUT char *a_rootVolNameP)
```

Description

[Opcode 151] Fetch the name of the volume which serves as the root of the AFS file system and place it into *a_rootVolNameP*. This name will always be less than `AFSNAMEMAX` characters long. Any *File Server* will respond to this call, not just the one hosting the root volume. The queried *File Server* first tries to discover the name of the root volume by reading from the `/usr/afs/etc/RootVolume` file on its local disks. If that file doesn't exist, then it will return the default value, namely "`root.afs`".

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`---` No error codes generated.

5.1.3.18 RXAFS_CheckToken — (Obsolete) Check that the given user identifier matches the one in the supplied authentication token

```
int RXAFS_CheckToken(IN struct rx_connection *a_rxConnP,
                     IN long ViceId,
                     IN AFSOpaque *token)
```

Description

[Opcode 152] This function only works for the now-obsolete RPC facility used by AFS, *R*. For modern systems using the *Rx* RPC mechanism, we always get an error return from this routine.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`ECONNREFUSED` Always returned on *Rx* connections.

5.1.3.19 RXAFS_GetTime — Get the *File Server*'s time of day

```
int RXAFS_GetTime(IN struct rx_connection *a_rxConnP,
                  OUT unsigned long *a_secondsP,
                  OUT unsigned long *a_uSecondsP)
```

Description

[Opcode 153] Get the current time of day from the *File Server* specified in the *Rx* connection information contained in *a_rxConnP*. The time is returned in elapsed seconds (*a_secondsP*) and microseconds (*a_uSecondsP*) since that standard UNIX "start of the world".

Error Codes

`---` No error codes generated.

5.1.3.20 RXAFS_NGetVolumeInfo — Get information about a volume given its name

```
int RXAFS_NGetVolumeInfo(IN struct rx_connection *a_rxConnP,
                         IN char *a_volNameP,
                         OUT AFSVolumeInfo *a_volInfoP)
```

Description

[Opcode 154] This function is identical to *RXAFS_GetVolumeInfo()* (see Section 5.1.3.14), except that it returns a `struct AFSVolumeInfo` instead of a `struct VolumeInfo`. The basic difference is that `struct AFSVolumeInfo` also carries an accompanying UDP port value for each *File Server* listed in the record.

5.1.3.21 RXAFS_BulkStatus — Fetch the status information regarding a set of given file system objects

```
int RXAFS_BulkStatus(IN struct rx_connection *a_rxConnP,
                     IN AFSCBFids *a_fidToStatArrayP,
                     OUT AFSBulkStats *a_currStatArrayP,
                     OUT AFSCBs *a_callBackArrayP,
                     OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 155] This routine is identical to *RXAFS_FetchStatus()* as described in Section 5.1.3.2, except for the fact that it allows the caller to ask for the current status fields for a set of up to `AFSCBMAX` (50) file identifiers at once.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` The number of file descriptors for which status information was requested is illegal.

5.1.3.22 RXAFS_SetLock — Set an advisory lock on the given file identifier

```
int RXAFS_SetLock(IN struct rx_connection *a_rxConnP,
                  IN AFSFid *a_fidToLockP,
                  IN ViceLockType a_lockType,
                  OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 156] Set an advisory lock on the file identified by a_fidToLockP. There are two types of locks that may be specified via *a_lockType*: `LockRead` and `LockWrite`. An advisory lock times out after `AFS_LOCKWAIT` (5) minutes, and must be extended in order to stay in force (see *RXAFS_ExtendLock()*, Section 5.1.3.23).

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EINVAL` An illegal lock type was specified.

`EWOULDBLOCK` The lock was already incompatibly granted to another party.

5.1.3.23 RXAFS_ExtendLock — Extend an advisory lock on a file

```
int RXAFS_ExtendLock(IN struct rx_connection *a_rxConnP,
                     IN AFSFid *a_fidToBeExtendedP,
                     OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 157] Extend the advisory lock that has already been granted to the caller on the file identified by *a_fidToBeExtendedP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EINVAL` The caller does not already have the given file locked.

5.1.3.24 RXAFS_ReleaseLock — Release the advisory lock on a file

```
int RXAFS_ReleaseLock(IN struct rx_connection *a_rxConnP,
                      IN AFSFid *a_fidToUnlockP,
                      OUT AFSVolSync *a_volSyncP)
```

Description

[Opcode 158] Release the advisory lock held on the file identified by *a_fidToUnlockP*. If this was the last lock on this file, the *File Server* will break all existing callbacks to this file.

Rx connection information for the related *File Server* is contained in *a_rxConnP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

5.1.3.25 RXAFS_XStatsVersion — Get the version number associated with the *File Server*'s extended statistics structure

```
int RXAFS_XStatsVersion(IN struct rx_connection *a_rxConnP,
                        OUT long *a_versionNumberP)
```

Description

[Opcode 159] This call asks the *File Server* for the current version number of the extended statistics structures it exports (see *RXAFS_GetXStats()*, Section 5.1.3.26). The version number is placed into *a_versionNumberP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

--- No error codes generated.

5.1.3.26 RXAFS_GetXStats — Get the current contents of the specified extended statistics structure

```
int RXAFS_GetXStats(IN struct rx_connection *a_rxConnP,
                    IN long a_clientVersionNumber,
                    IN long a_collectionNumber,
                    OUT long *a_srvVersionNumberP,
                    OUT long *a_timeP,
                    OUT AFS_CollData *a_dataP)
```

Description

[Opcode 160] This function fetches the contents of the specified *File Server* extended statistics structure. The caller provides the version number of the data it expects to receive in *a_clientVersionNumber*. Also provided in *a_collectionNumber* is the numerical identifier for the desired **data collection**. There are currently three of these data collections defined: `AFS_XSTATSCOLL_CALL_INFO`, which is the list of tallies of the number of invocations of internal *File Server* procedure calls, `AFS_XSTATSCOLL_PERF_INFO`, which is a list of performance-related numbers, and `AFS_XSTATSCOLL_FULL_PERF_INFO`, which is an extended list of performance-related numbers. The precise contents of these collections are described in Sections 5.1.2.7 and 5.1.2.11. The current version number of the *File Server* collections is returned in *a_srvVersionNumberP*, and is always set upon return, even if the caller has asked for a different version. If the correct version number has been specified, and a supported collection number given, then the collection data is returned in *a_dataP*. The time of collection is also returned, being placed in *a_timeP*.

Rx connection information for the related *File Server* is contained in *a_rxConnP*.

Error Codes

`---` No error codes are generated.

5.1.4 Streamed Function Calls

There are two *streamed* functions in the *File Server* RPC interface, used to fetch and store arbitrary amounts of data from a file. While some non-streamed calls pass such variable-length objects as `struct AFSCBFids`, these objects have a pre-determined maximum size.

The two streamed RPC functions are also distinctive in that their single *Rxgen* declarations generate not one but *two* client-side stub routines. The first is used to ship the `IN` parameters off to the designated *File Server*, and the second to gather the `OUT` parameters and the error code. If a streamed definition declares a routine named *X_YZ()*, the two resulting stubs will be named *StartX_YZ()* and *EndX_YZ()*. It is the application programmer's job to first invoke *StartX_YZ()*, then manage the unbounded data transfer, then finish up by calling *EndX_YZ()*. The first longword in the unbounded data stream being fetched from a *File Server* contains the number of data bytes to follow. The application then reads the specified number of bytes from the stream.

The following sections describe the four client-side functions resulting from the *FetchData()* and *StoreData()* declarations in the *Rxgen* interface definition file. These are the actual routines the application programmer will include in the client code. For reference, here are the interface definitions that generate these functions. Note that the `split` keyword is what causes *Rxgen* to generate the separate start and end routines. In each case, the number after the equal sign specifies the function's identifying opcode number. The opcode is passed to the *File Server* by the *StartRXAFS_FetchData()* and *StartRXAFS_StoreData()* stub routines.

```
FetchData(IN  AFSFid *a_fidToFetchP,
          IN  long a_offset,
          IN  long a_lenInBytes,
          OUT AFSFetchStatus *a_fidStatP,
          OUT AFSCallBack *a_callBackP,
          OUT AFSVolSync *a_volSyncP) split = 130;
```

```
StoreData(IN  AFSFid *Fid,
          IN  AFSStoreStatus *InStatus,
          IN  long Pos,
          IN  long Length,
          IN  long FileLength,
          OUT AFSFetchStatus *OutStatus,
          OUT AFSVolSync *a_volSyncP) split = 133;
```

5.1.4.1 StartRXAFS_FetchData — Begin a request to fetch file data

```
int StartRXAFS_FetchData(IN struct rx_call *a_rxCallP,
                         IN AFSFid *a_fidToFetchP,
                         IN long a_offset,
                         IN long a_lenInBytes)
```

Description

Begin a request for *a_lenInBytes* bytes of data starting at byte offset *a_offset* from the file identified by *a_fidToFetchP*. After successful completion of this call, the data stream will make the desired bytes accessible. The first longword in the stream contains the number of bytes to actually follow.

Rx call information to the related *File Server* is contained in *a_rxCallP*.

Error Codes

--- No error codes generated.

5.1.4.2 EndRXAFS_FetchData — Conclude a request to fetch file data

```
int EndRXAFS_FetchData(IN struct rx_call *a_rxCallP,
                       OUT AFSFetchStatus *a_fidStatP,
                       OUT AFSCallBack *a_callBackP,
                       OUT AFSVolSync *a_volSyncP)
```

Description

Conclude a request to fetch file data, as commenced by an *StartRXAFS_FetchData()* invocation. By the time this routine has been called, all of the desired data has been read off the data stream. The status fields for the file from which the data was read are stored in *a_fidStatP*. If the file was from a read/write volume, its callback information is placed in *a_callBackP*.

Rx call information to the related *File Server* is contained in *a_rxCallP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

- `EACCES` The caller does not have the necessary access rights.
- `EIO` Given file could not be opened or statted on the *File Server*, or there was an error reading the given data off the *File Server*'s disk.
- `-31` An *Rx* write into the stream ended prematurely.

5.1.4.3 StartRXAFS_StoreData — Begin a request to store file data

```
int StartRXAFS_StoreData(IN struct rx_call *a_rxCallP,
                         IN AFSFid *a_fidToStoreP,
                         IN AFSStoreStatus *a_fidStatusP,
                         IN long a_offset,
                         IN long a_lenInBytes,
                         IN long a_fileLenInBytes)
```

Description

Begin a request to write *a_lenInBytes* of data starting at byte offset *a_offset* to the file identified by *a_fidToStoreP*, causing that file's length to become *a_fileLenInBytes* bytes. After successful completion of this call, the data stream will be ready to begin accepting the actual data being written.

Rx call information to the related *File Server* is contained in *a_rxCallP*.

Error Codes

`---` No error codes generated.

5.1.4.4 **EndRXAFS_StoreData** — Conclude a request to store file data

```
int EndRXAFS_StoreData(IN struct rx_call *a_rxCallP,
                       OUT AFSFetchStatus *a_fidStatP,
                       OUT AFSCallBack *a_callBackP,
                       OUT AFSVolSync *a_volSyncP)
```

Description

Conclude a request to store file data, as commenced by a *StartRXAFS_StoreData()* invocation. By the time this routine has been called, all of the file data has been inserted into the data stream. The status fields for the file to which the data was written are stored in *a_fidStatP*. All existing callbacks to the given file are broken before the store concludes.

Rx call information to the related *File Server* is contained in *a_rxCallP*. Volume version information is returned for synchronization purposes in *a_volSyncP*.

Error Codes

`EACCES` The caller does not have the necessary access rights.

`EISDIR` The file being written to is a symbolic link.

`ENOSPEC` A write to the *File Server*'s file on local disk failed.

`-32` A short read was encountered by the *File Server* on the data stream.

5.1.5 Example of Streamed Function Call Usage

5.1.5.1 Preface

The following code fragment is offered as an example of how to use the streamed *File Server* RPC calls. In this case, a client fetches some amount of data from the given *File Server* and writes it to a local file it uses to cache the information. For simplicity, many issues faced by a true application programmer are *not* addressed here. These issues include locking, managing file chunking, data version number mismatches, volume location, *Rx* connection management, defensive programming (e.g., checking parameters before using them), client-side cache management algorithms, callback management, and full error detection and recovery. Pseudocode is incorporated when appropriate to keep the level of detail reasonable. For further descriptions of some of these details and issues, the reader is referred to such companion documents as *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*, *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*, and *AFS-3 Programmer's Reference: Architectural Overview.*

A discussion of the methods used within the example code fragment follows immediately afterwards in Section 5.1.5.3.

5.1.5.2 Code Fragment Illustrating Fetch Operation

```
int code;                  /*Return code*/
long bytesRead;            /*Num bytes read from Rx*/
struct myConnInfo *connP;  /*Includes Rx conn info*/
struct rx_call *rxCallP;   /*Rx call ptr*/
struct AFSFid *afsFidP;    /*Fid for file to fetch*/
int lclFid;                /*Fid for local cache file*/
long offsetBytes;          /*Starting fetch offset*/
long bytesToFetch;         /*Num bytes to fetch*/
long bytesFromFS;          /*Num bytes FileServer returns*/
char *fetchBuffP;          /*Buffer to hold stream data*/
int currReadBytes;         /*Num bytes for current read*/

    /*
     * Assume that connP, afsFidP, offsetBytes, lclFid,and
     * bytesToFetch have all been given their desired values.
     */
        .
        .
        .
    rxCallP = rx_NewCall(connP->rxConnP);
    code = StartRXAFS_FetchData(
                rxCallP,        /*Rx call to use*/
```

```
              afsFidP,        /*Fid being fetched from*/
              offsetBytes,    /*Offset in bytes*/
              bytesToFetch);  /*Num bytes wanted*/
    if (code == 0) {
        bytesRead = rx_Read(rxCallP, &bytesFromFS, sizeof(long));
        if (bytesRead != sizeof(long))
            ExitWithError(SHORT_RX_READ);
        bytesFromFS = ntohl(bytesFromFS);
        xmitBuffer = malloc(FETCH_BUFF_BYTES);
        lclFid = open(CacheFileName, O_RDWR, mode);
        pos = lseek(lclFid, offsetBytes, L_SET);
        while (bytesToFetch > 0) {
            currReadBytes =
                (bytesToFetch > FETCH_BUFF_BYTES) ?
                    FETCH_BUFF_BYTES : bytesToFetch;
            bytesRead = rx_Read(rxCallP, fetchBuffP, currReadBytes);
            if (bytesRead != currReadBytes)
                ExitWithError(SHORT_RX_READ);
            code = write(lclFid, fetchBuffP, currReadBytes);
            if (code)
                ExitWithError(LCL_WRITE_FAILED);
            bytesToFetch -= bytesRead;
        } /*Read from the Rx stream*/
        close(lclFid);
    }
    else
        ExitWithError(code);
    code = EndRXAFS_FetchData(
              rxCallP,        /*Rx call to use*/
              fidStatP,       /*Resulting stat fields*/
              fidCallBackP,   /*Resulting callback info*/
              volSynchP);     /*Resulting volume sync info*/
    code = rx_EndCall(rxCallP, code);
    return(code);
        .
        .
        .
```

5.1.5.3 Discussion and Analysis

The opening assumption in this discussion is that all the information required to do the fetch has already been set up. These mandatory variables are the client-side connection information for the *File Server* hosting the desired file, the corresponding AFS file identifier, the byte offset into the file, the number of bytes to fetch, and the identifier for the local file serving as a cached copy.

Given the *Rx* connection information stored in the client's `connP` record, *rx_NewCall()* is used to create a new *Rx* call to handle this fetch operation. The structure containing this call handle is placed into `rxCallP`. This call handle is used immediately in the invocation of *StartRXAFS_FetchData()*. If this setup call fails, the fragment exits. Upon success, though, the *File Server* will commence writing the desired data into the *Rx* data stream. The *File Server* first writes a single longword onto the stream announcing to the client how many bytes of data will actually follow. The fragment reads this number with its first *rx_Read()* call. Since all *Rx* stream data is written in network byte order, the fragment translates the byte count to its own host byte order first to properly interpret it. Once the number of bytes to appear on the stream is known, the client code proceeds to open the appropriate cache file on its own local disk and seeks to the appropriate spot within it. A buffer into which the stream data will be placed is also created at this time.

The example code then falls into a loop where it reads all of the data from the *File Server* and stores it in the corresponding place in the local cache file. For each iteration, the code decides whether to read a full buffer's worth or the remaining number of bytes, whichever is smaller. After all the data is pulled off the *Rx* stream, the local cache file is closed. At this point, the example finishes off the RPC by calling *EndRXAFS_FetchData()*. This gathers in the required set of `OUT` parameters, namely the status fields for the file just fetched, callback and volume synchronization information, and the overall error code for the streamed routine. The *Rx* call created to perform the fetch is then terminated and cleaned up by invoking *rx_EndCall()*.

5.1.6 Required Caller Functionality

The AFS *File Server* RPC interface was originally designed to interact only with *Cache Manager* agents, and thus made some assumptions about its callers. In particular, the *File Server* expected that the agents calling it would potentially have stored callback state on file system objects, and would have to be periodically pinged in order to garbage-collect its records, removing information on dead client machines. Thus, any entity making direct calls to this interface **must** mimic certain *Cache Manager* actions, and respond to certain *Cache Manager* RPC interface calls.

To be safe, any application calling the *File Server* RPC interface directly should export the entire *Cache Manager* RPC interface. Realistically, though, it will only need to provide stubs for the three calls from this interface that *File Server*s know how to make: *RXAFSCB_InitCallBackState()*, *RXAFSCB_Probe()* and *RXAFSCB_CallBack()*. The very first *File Server* call made by this application will prompt the given *File Server* to call *RXAFSCB_InitCallBackState()*. This informs the application that the *File Server* has no record of its existence and hence this "*Cache Manager*" should clear all callback information for that server. Once the application responds positively to the inital *RXAFSCB_InitCallBackState()*, the *File Server* will treat it as a bona fide, fully-fledged

Cache Manager, and probe it every so often with *RXAFSCB_Probe()* calls to make sure it is still alive.

5.2 Signal Interface

While the majority of communication with AFS *File Server*s occurs over the RPC interface, some important operations are invoked by sending UNIX signals to the process. This section describes the set of signals recognized by the *File Server* and the actions they trigger upon receipt, as summarized below:

- `SIGQUIT`: Shut down a *File Server*.
- `SIGTSTP`: Upgrade debugging output level.
- `SIGHUP`: Reset debugging output level.
- `SIGTERM`: Generate debugging output specifically concerning open files within the *File Server* process.

5.2.1 `SIGQUIT`: Server Shutdown

Upon receipt of this signal, the *File Server* shuts itself down in an orderly fashion. It first writes a message to the console and to its log file (*/usr/afs/logs/FileLog*) stating that a shutdown has commenced. The *File Server* then flushes all modified buffers and prints out a set of internal statistics, including cache and disk numbers. Finally, each attached volume is taken offline, which means the volume header is written to disk with the appropriate bits set.

In typical usage, human operators do not send the `SIGQUIT` signal directly to the *File Server* in order to affect an orderly shutdown. Rather, the *BOS Server* managing the server processes on that machine issues the signal upon receipt of a properly-authorized shutdown RPC request.

5.2.2 `SIGTSTP`: Upgrade Debugging Level

Arrival of a `SIGTSTP` signal results in an increase of the debugging level used by the *File Server*. The routines used for writing to log files are sensitive to this debugging level, as recorded in the global `LogLevel` variable. Specifically, these routines will only generate output if the value of `LogLevel` is greater than or equal to the value of its threshold parameter. By default, the *File Server* sets `LogLevel` to zero upon startup. If a `SIGTSTP` signal is received when the debugging level is zero, it will be bumped to 1. If the signal arrives when the debugging level is positive, its value will be multiplied by 5. Thus, as more `SIGTSTP`s are received, the set of debugging messages eligible to be delivered to log files grows.

Since the `SIGTSTP` signal is not supported under IBM's AIX 2.2.1 operating system, this form of debugging output manipulation is not possible on those platforms.

5.2.3 `SIGHUP`: Reset Debugging Level

Receiving a `SIGHUP` signal causes a *File Server* to reset its debugging level to zero. This effectively reduces the set of debugging messages eligible for delivery to log files to a bare minimum. This signal is used in conjunction with `SIGTSTP` to manage the verbosity of log information.

Since the `SIGHUP` signal is not supported under IBM's AIX 2.2.1 operating system, this form of debugging output manipulation is not possible on those platforms.

5.2.4 `SIGTERM`: File Descriptor Check

Receipt of a `SIGTERM` signal triggers a routine which sweeps through the given *File Server*'s UNIX file descriptors. For each possible UNIX fid slot, an *fstat()* is performed on that descriptor, and the particulars of each open file are printed out. This action is designed solely for debugging purposes.

5.3 Command Line Interface

Another interface exported by the *File Server* is the set of command line switches it accepts. Using these switches, many server parameters and actions can be set. Under normal conditions, the *File Server* process is started up by the *BOS Server* on that machine, as described in *AFS-3 Programmer's Reference: BOS Server Interface*. So, in order to utilize any combination of these command-line options, the system administrator must define the *File Server* bnode in such a way that these parameters are properly included. Note that the switch names must be typed exactly as listed, and that abbreviations are not allowed. Thus, specifying **-b 300** on the command line is unambiguous, directing that 300 buffers are to be allocated. It is *not* an abbreviation for the **-banner** switch, asking that a message is to be printed to the console periodically.

A description of the set of currently-supported command line switches follows.

- **-b** <*# buffers*> Choose the number of 2,048-byte data buffers to allocate at system startup. If this switch is not provided, the *File Server* will operate with 70 such buffers by default.
- **-banner** This switch instructs the *File Server* to print messages to the console every 10 minutes to demonstrate it is still running correctly. The text of the printed message is: `File Server is running at` <*time*>.
- **-cb** <*# callbacks stored*> Specify the maximum number of callback records stored simultaneously by the *File Server*. The default pool size is 20,000 records.
- **-d** <*debug level*> Set the debugging output level at which *File Server* runs to the value provided. Specifically, the `LogLevel` global variable is set to the given value (See Section 5.2.2). If this switch is not provided, the default initial *File Server* debugging level is set to zero, producing the minimal debugging output to the log files.
- **-k** <*stack size*> Set the stack size to provide server LWPs upon creation, measured in 1,024-byte blocks. The default LWP stack size is 24 blocks, or 24,576 bytes.
- **-l** <*large (directory) vnodes*> Select the number of "large" vnodes the *File Server* will cache. These vnodes are suitable for recording information about AFS directories. The extra space in the vnode allows ACL information to be stored along with the directory. The default allocation value is 200 directory vnodes.
- **-pctspare** <*percent overrun blocks past quota*> Similar to the **-spare** switch, except that the number of allowable overrun blocks is expressed as a percentage of the given volume's quota. Note: this switch cannot be used in combination with the **-spare** switch.
- **-rxdbg** Instruct the *File Server* to open a file named `rx_dbg` in the current directory, into which the *Rx* package will write general debugging information. If the file is already open (due to the appearance of the **-rxdbge** switch earlier in the command line), this results in a no-op.
- **-rxdbge** Instruct the *File Server* to open a file named `rx_dbg` in the current directory, into which the *Rx* package will write debugging information related to its event-scheduling activities. If the file is already open (due to the appearance of the **-rxdbg** switch earlier in the command line), this results in a no-op.

- **-rxpck** <*# packets*> Set the number of extra *Rx* packet buffers to hold in reserve. These pre-allocated buffers assist in responding to spikes in network traffic demands. By default, 100 such packet buffers are maintained.

- **-s** <*small (file) vnodes*> Select the number of "small" vnodes the *File Server* will cache. These vnodes are suitable for recording information about non-directory files. As with directory vnodes, the *File Server* will allocate 200 small vnodes by default.

- **-spare** <*# overrun blocks to allow*> Tell the *File Server* to allow users performing a store operation to overrun the host volume's disk quota by a certain number of (1,024-byte) blocks. In other words, the first store resulting in a quota overrun will be allowed to succeed if and only if it uses no more than these many blocks beyond the quota. Further store operations will be rejected until the volume's storage is once again reduced below quota. By default, overruns of 1,024 blocks of 1,024 bytes each (1 megabyte total) are tolerated. Note: this switch cannot be used in combination with the **-pctspare** switch.

- **-w** <*callback wait interval in seconds*> This switch determines how often the *File Server* periodic daemon lightweight processes run. Among other things, these daemon LWPs check on the validity of callback records, keep disk usage statistics up to date, and check the health of the various client machines that have previously interacted with the *File Server*. For a full description of these daemon LWPs, consult Section 2.3. The associated argument specifies the number of seconds to sleep between daemon invocations. By default, these periodic daemons run every 300 seconds (5 minutes).

Chapter 6

Cache Manager **Interfaces**

6.1 Overview

There are several interfaces offered by the *Cache Manager*, allowing clients to access the files stored by the community of AFS *File Server*s, to configure the *Cache Manager*'s behavior and resources, to store and retrieve authentication information, to specify the location of community *Authentication Server* and *Volume Location Server* services, and to observe and debug the *Cache Manager*'s state and actions. This chapter will cover the following five interfaces to the *Cache Manager*:

- **ioctl()**: The standard UNIX **ioctl()** system call has been extended to include more operations, namely waiting until data stores to a *File Server* complete before returning to the caller (`VIOCCLOSEWAIT`) and getting the name of the cell in which an open file resides (`VIOCIGETCELL`).
- **pioctl()**: An additional system call is provided through which applications can access operations specific to AFS, which are often tied to a particular pathname. These operations include Access Control List (ACL) and mount point management, Kerberos ticket management, cache configuration, cell configuration, and status of *File Server*s.
- **RPC**: Interface by which outside servers and investigators can manipulate the *Cache Manager*. There are two main categories of routines: callback management, typically called by the *File Server*, and debugging/statistics, called by programs such as *cmdebug* and via the *xstat* user-level library for collection of extended statistics.

- **Files**: Much of the *Cache Manager*'s configuration information, as well as its view of the AFS services available from the outside world, is obtained from parsing various files. One set of these files is typically located in */usr/vice/etc*, and includes *CellServDB*, *ThisCell*, and *cacheinfo*. Another set is usually found in */usr/vice/cache*, namely *CacheItems*, *VolumeItems*, and *AFSLog*.

- **Mariner**: This is the interface by which file transfer activity between the *Cache Manager* and *File Server*s may be monitored. Specifically, it is used to monitor the names of the files and directories being fetched and/or stored over the network.

Another important component not described in this document is the *afsd* program. It is *afsd*'s job to initialize the *Cache Manager* on a given machine and to start up its related daemon threads. It accepts a host of configuration decisions via its command-line interface. In addition, it parses some of the information kept in the configuration files mentioned above and passes that information to the *Cache Manager*. The reader may find a full description of *afsd* in the *AFS 3.0 Command Reference Manual*[2].

6.2 Definitions

This section defines data structures that are used by the **pioctl()** calls.

6.2.1 struct VenusFid

The *Cache Manager* is the sole active AFS agent aware of the cellular architecture of the system. Since AFS file identifiers are not guaranteed to be unique across cell boundaries, it must further qualify them for its own internal bookkeeping. The `struct VenusFid` provides just such additional qualification, attaching the *Cache Manager*'s internal cell identifier to the standard AFS fid.

Fields

`long Cell` - The internal identifier for the cell in which the file resides.

`struct ViceFid Fid` - The AFS file identifier within the above cell.

6.2.2 struct ClearToken

This is the clear-text version of an AFS token of identity. Its fields are encrypted into the secret token format, and are made easily available to the *Cache Manager* in this structure.

Fields

`long AuthHandle` - Key version number.

`char HandShakeKey[8]` - Session key.

`long ViceId` - Identifier for the AFS principal represented by this token.

`long BeginTimestamp` - Timestamp of when this token was minted, and hence came into effect.

`long EndTimestamp` - Timestamp of when this token is considered to be expired, and thus disregarded.

6.3 ioctl() Interface

The standard UNIX **ioctl()** system call performs operations on file system objects referenced with an open file descriptor. AFS has augmented this system call with two additional operations, one to perform "safe stores", and one to get the name of the cell in which a file resides. A third **ioctl()** extension is now obsolete, namely aborting a store operation currently in progress.

6.3.1 VIOCCLOSEWAIT

[Opcode 1] Normally, a client performing a UNIX *close()* call on an AFS file resumes once the store operation on the given file data to the host *File Server* has commenced but before it has completed. Thus, it is possible that the store could actually fail (say, because of network partition or server crashes) without the client's knowledge. This new ioctl opcode specifies to the *Cache Manager* that all future *close()* operations will wait until the associated store operation to the *File Server* has completed fully before returning.

6.3.2 VIOCABORT

[Opcode 2] This **ioctl()** extension is now obsolete. This call results in a noop.

The original intention of this call was to allow a store operation currently in progress to a *File Server* on the named fid to be aborted.

6.3.3 VIOIGETCELL

[Opcode 3] Get the name of the cell in which the given fid resides. If the file is not an AFS file, then `ENOTTY` is returned. The output buffer specified in the data area must be large enough to hold the null-terminated string representing the file's cell, otherwise `EFAULT` is returned. However, an `out_size` value of zero specifies that the cell name is *not* to be copied into the output buffer. In this case, the caller is simply interested in whether the file is in AFS, and not its exact cell of residence.

6.4 pioctl() Interface

6.4.1 Introduction

There is a new UNIX system call, **pioctl()**, which has been defined especially to support the AFS *Cache Manager*. Its functional definition is as follows:

```
int afs_syscall_pioctl(IN char *a_pathP,
                       IN int a_opcode,
                       IN struct ViceIoctl *a_paramsP,
                       IN int a_followSymLinks)
```

This new call is much like the standard **ioctl()** call, but differs in that the affected file (when applicable) is specified by its path, not by a file descriptor. Another difference is the fourth parameter, `a_followSymLinks`, determines which file should be used should *a_pathP* be a symbolic link. If *a_followSymLinks* be set to 1, then the symbolic link is followed to its target, and the **pioctl()** is applied to that resulting file. If *a_followSymLinks* is set to 0, then the **pioctl()** applies to the symbolic link itself.

Not all **pioctl()** calls affect files. In those cases, the *a_pathP* parameter should be set to a null pointer. The second parameter to **pioctl()**, *a_opcode*, specifies which operation is to be performed. The opcode for each of these operations is included in the text of the description. Note that not all **pioctl()** opcodes are in use. These unused values correspond to obsolete operations.

The descriptions that follow identify some of the possible error codes for each **pioctl()** opcode, but do not offer a comprehensive lists. All **pioctl()** calls return 0 upon success.

The rest of this section proceeds to describe the individual opcodes available. First, though, one asymmetry in this opcode set is pointed out, namely that while various operations are defined on AFS mount points, there is no direct way to *create* a mount point.

This documentation partitions the **pioctl()** into several groups:

- **Volume operations**
- *File Server* **operations**
- **Cell Operations**
- **Authentication Operations**
- **ACL Operations**
- **Cache operations**
- **Miscellaneous operations**

For all **pioctl()**s, the fields within the *a_paramsP* parameter will be referred to directly. Thus, the values of *in*, *in_size*, *out*, and *out_size* are discussed, rather than the settings for *a_paramsP->in*, *a_paramsP->in_size*, *a_paramsP->out*, and *a_paramsP->out_size*.

For convenience of reference, a list of the actively-supported **pioctl()**s, their opcodes, and brief description appears (in opcode order) below.

- *[1]* `VIOCSETAL` : Set the ACL on a directory
- *[2]* `VIOCGETAL` : Get the ACL for a directory
- *[3]* `VIOCSETTOK` : Set the caller's token for a cell
- *[4]* `VIOCGETVOLSTAT` : Get volume status

- *[5]* VIOCSETVOLSTAT : Set volume status
- *[6]* VIOCFLUSH : Flush an object from the cache
- *[8]* VIOCGETTOK : Get the caller's token for a cell
- *[9]* VIOCUNLOG : Discard authentication information
- *[10]* VIOCCKSERV : Check the status of one or more *File Server*s
- *[11]* VIOCCKBACK : Mark cached volume info as stale
- *[12]* VIOCCKCONN : Check caller's tokens/connections
- *[14]* VIOCWHEREIS : Find host(s) for a volume
- *[20]* VIOCACCESS : Check caller's access on object
- *[21]* VIOCUNPAG : See *[9]* VIOCUNLOG
- *[22]* VIOCGETFID : Get fid for named object
- *[24]* VIOCSETCACHESIZE : Set maximum cache size in blocks
- *[25]* VIOCFLUSHCB : Unilaterally drop a callback
- *[26]* VIOCNEWCELL : Set cell service information
- *[27]* VIOCGETCELL : Get cell configuration entry
- *[28]* VIOCAFS_DELETE_MT_PT : Delete a mount point
- *[29]* VIOC_AFS_STAT_MT_PT : Get the contents of a mount point
- *[30]* VIOC_FILE_CELL_NAME : Get cell hosting a given object
- *[31]* VIOC_GET_WS_CELL : Get caller's home cell name
- *[32]* VIOC_AFS_MARINER_HOST : Get/set file transfer monitoring output
- *[33]* VIOC_GET_PRIMARY_CELL : Get the caller's primary cell
- *[34]* VIOC_VENUSLOG : Enable/disable *Cache Manager* logging
- *[35]* VIOC_GETCELLSTATUS : Get status info for a cell entry
- *[36]* VIOC_SETCELLSTATUS : Set status info for a cell entry
- *[37]* VIOC_FLUSHVOLUME : Flush cached data from a volume

- *[38]* `VIOC_AFS_SYSNAME` : Get/set the @sys mapping
- *[39]* `VIOC_EXPORTAFS` : Enable/disable NFS/AFS translation
- *[40]* `VIOCGETCACHEPARAMS` : Get current cache parameter values

6.4.2 Mount Point Asymmetry

There is an irregularity which deserves to be mentioned regarding the **pioctl()** interface. There are **pioctl()** operations for getting information about a mount point (`VIOC_AFS_STAT_MT_PT`) and for deleting a mount point (`VIOC_AFS_DELETE_MT_PT`), but no operation for creating mount points. To create a mount point, a symbolic link obeying a particular format must be created. The first character must be either a "%" or a "#", depending on the type of mount point being created (see the discussion in Section 6.4.4.4). If the mount point carries the name of the cell explicitly, the full cell name will appear next, followed by a colon. In all cases, the next portion of the mount point is the volume name. By convention, the last character of a mount point must always be a period ("."). This trailing period is not visible in the output from `fs lsmount`.

6.4.3 Volume Operations

There are several **pioctl()** opcodes dealing with AFS volumes. It is possible to get and set volume information (`VIOCGETVOLSTAT, VIOCSETVOLSTAT`), discover which volume hosts a particular file system object (`VIOCWHEREIS`), remove all objects cached from a given volume (`VIOC_FLUSHVOLUME`), and revalidate cached volume information (`VIOCCKBACK`).

6.4.3.1 `VIOCGETVOLSTAT`: Get volume status for pathname

[Opcode 4] Fetch information concerning the volume that contains the file system object named by *a_pathP*. There is no other input for this call, so *in_size* should be set to zero. The status information is placed into the buffer named by *out*, if *out_size* is set to a value of `sizeof(struct VolumeStatus)` or larger. Included in the volume information are the volume's ID, quota, and number of blocks used in the volume as well as the disk partition on which it resides. Internally, the *Cache Manager* calls the *RXAFS_GetVolumeInfo()* RPC (See Section 5.1.3.14) to fetch the volume status.

Among the possible error returns, `EINVAL` indicates that the object named by *a_pathP* could not be found.

6.4.3.2 `VIOCSETVOLSTAT`: Set volume status for pathname

[Opcode 5] Set the status fields for the volume hosting the file system object named by *a_pathP*. The first object placed into the input buffer *in* is the new status image. Only those fields that may change, namely *MinQuota* and *MaxQuota* fields, are interpreted upon receipt by the *File Server*, and are set to the desired values. Immediately after the `struct VolumeStatus` image, the caller must place the null-terminated string name of the volume involved in the input buffer. New settings for the offline message and MOTD (Message of the Day) strings may appear after the volume name. If there are no changes in the offline and/or MOTD messages, a null string must appear for that item. The *in_size* parameter must be set to the total number of bytes so inserted, including the nulls after each string. Internally, the *Cache Manager* calls the *RXAFS_SetVolumeStatus()* RPC (See Section 5.1.3.16) to store the new volume status.

Among the possible error returns, `EINVAL` indicates that the object named by *a_pathP* could not be found.

6.4.3.3 `VIOCWHEREIS`: Find the server(s) hosting the pathname's volume

[Opcode 14] Find the set of machines that host the volume in which the file system object named by *a_pathP* resides. The input buffer *in* is not used by this call, so *in_size* should be set to zero. The output buffer indicated by *out* is filled with up to 8 IP addresses, one for each *File Server* hosting the indicated volume. Thus, *out_size* should be set to at least `(8*sizeof(long))`. This group of hosts is terminated by the first zeroed IP address that appears in the list, but under no circumstances are more than 8 host IP addresses returned.

Among the possible error returns is `EINVAL`, indicating that the pathname is not in AFS, hence is not contained within a volume. If `ENODEV` is returned, the associated volume information could not be obtained.

6.4.3.4 `VIOC_FLUSHVOLUME`: Flush all data cached from the pathname's volume

[Opcode 37] Determine the volume in which the file system object named by *a_pathP* resides, and then throw away all currently cached copies of files that the *Cache Manager* has obtained from that volume. This call is typically used should a user suspect there is some cache corruption associated with the files from a given volume.

6.4.3.5 `VIOCCKBACK`: Check validity of all cached volume information

[Opcode 11] Ask the *Cache Manager* to check the validity of all cached volume information. None of the call's parameters are referenced in this call, so *a_pathP* and *in* should be set to the null pointer, and *in_size* and *out_size* should be set to zero.

This operation is performed in two steps:

1. The *Cache Manager* first refreshes its knowledge of the root volume, usually named `root.afs`. On success, it wakes up any of its own threads waiting on the arrival of this information, should it have been previously unreachable. This typically happens should the *Cache Manager* discover in its startup sequence that information on the root volume is unavailable. Lacking this knowledge at startup time, the *Cache Manager* settles into a semi-quiescent state, checking every so often to see if volume service is available and thus may complete its own initialization.
2. Each cached volume record is flagged as being stale. Any future attempt to access information from these volumes will result in the volume record's data first being refreshed from the *Volume Location Server*.

6.4.4 *File Server* Operations

One group of **pioctl()** opcodes is aimed at performing operations against one or more *File Server*s directly. Specifically, a caller may translate a pathname into the corresponding AFS fid (`VIOCGETFID`), unilaterally discard a set of callback promises (`VIOCFLUSHCB`), get status on mount points (`VIOC_AFS_STAT_MT_PT`), delete unwanted mount points (`VIOC_AFS_DELETE_MT_PT`), and check the health of a group of *File Server*s (`VIOCCKSERV`).

6.4.4.1 `VIOCGETFID`: Get augmented fid for named file system object

[Opcode 22] Return the augmented file identifier for the file system object named by *a_pathP*. The desired `struct VenusFid` is placed in the output buffer specified by *out*. The output buffer size, as indicated by the *out_size* parameter, must be set to the value of `sizeof(struct VenusFid)` or greater. The input buffer is not referenced in this call, so *in* should be set to the null pointer and *in_size* set to zero.

Among the possible error returns, `EINVAL` indicates that the object named by *a_pathP* was not found.

6.4.4.2 `VIOCFLUSHCB`: Unilaterally drop a callback

[Opcode 25] Remove any callback information kept by the *Cache Manager* on the file system object named by *a_pathP*. Internally, the *Cache Manager* executes a call to the *RXAFS_GiveUpCallBacks()* RPC (See Section 5.1.3.13) to inform the appropriate *File Server* that it is being released from its particular callback promise. Note that if the named file resides on a read-only volume, then the above call is not made, and success is returned immediately. This optimization is possible because AFS *File Server*s do not grant callbacks on files from read-only volumes.

Among the possible error returns is `EINVAL`, which indicates that the object named by *a_pathP* was not found.

6.4.4.3 `VIOC_AFS_DELETE_MT_PT`: Delete a mount point

[Opcode 28] Remove an AFS mount point. The name of the directory in which the mount point exists is specified by *a_pathP*, and the string name of the mount point within this directory is provided through the *in* parameter. The input buffer length, *in_size*, is set to the length of the mount point name itself, including the trailing null. The output buffer is not accessed by this call, so *out* should be set to the null pointer and *out_size* to zero.

One important note is that the *a_followSymLinks* argument **must be set to zero for correct operation**. This is counter-intuitive, since at first glance it seems that a symbolic link that resolves to a directory should be a valid pathname parameter. However, recall that mount points are implemented as symbolic links that do not actually point to another file system object, but rather simply contain cell and volume information (see the description in Section 6.4.2). This "special" symbolic link must not be resolved by the **pioctl()**, but rather presented as-is to the *Cache Manager*, which then properly interprets it and generates a reference to the given volume's root directory. As an unfortunate side-effect, a perfectly valid symbolic link referring to a directory will be rejected out of hand by this operation as a value for the *a_pathP* parameter.

Among the possible error returns, `EINVAL` reports that the named directory was not found, and `ENOTDIR` indicates that the pathname contained within *a_pathP* is not a directory.

6.4.4.4 `VIOC_AFS_STAT_MT_PT`: Get the contents of a mount point

[Opcode 29] Return the contents of the given mount point. The directory in which the mount point in question resides is provided via the *a_pathP* argument, and the *in* buffer

contains the name of the mount point object within this directory. As usual, *in_size* is set to the length of the input buffer, including the trailing null. If the given object is truly a mount point and the *out* buffer is large enough (its length appears in *out_size*), the mount point's contents are stored into *out*.

The mount point string returned obeys a stylized format, as fully described in Section 5.6.2 of the *AFS 3.0 System Administrator's Guide*[1]. Briefly, a leading pound sign ("#") indicates a standard mount point, inheriting the read-only or read-write preferences of the mount point's containing volume. On the other hand, a leading percent sign ("%") advises the *Cache Manager* to cross into the read-write version of the volume, regardless of the existence of read-only clones. If a colon (":") separator occurs, the portion up to the colon itself denotes the fully-qualified cell name hosting the volume. The rest of the string is the volume name itself.

Among the possible error codes is `EINVAL`, indicating that the named object is not an AFS mount point. Should the name passed in *a_pathP* be something other than a directory, then `ENOTDIR` is returned.

6.4.4.5 `VIOCCKSERV`: Check the status of one or more *File Server*s

[Opcode 10] Check the status of the *File Server*s that have been contacted over the lifetime of the *Cache Manager*. The *a_pathP* parameter is ignored by this call, so it should be set to the null pointer. The input parameters as specified by *in* are completely optional. If something is placed in the input buffer, namely *in_size* is not zero, then the first item stored there is a longword used as a bit array of flags. These flags carry instructions as to the domain and the "thoroughness" of this check.

Only the settings of the least-significant two bits are recognized. Enabling the lowest bit tells the *Cache Manager* not to ping its list of servers, but simply report their status as contained in the internal server records. Enabling the next-higher bit limits the search to only those *File Server*s in a given cell. If *in_size* is greater than `sizeof(long)`, a null-terminated cell name string follows the initial flag array, specifying the cell to check. If this search bit is set but no cell name string follows the longword of flags, then the search is restricted to those servers contacted from the same cell as the caller.

This call returns at least one longword into the output buffer *out*, specifying the number of hosts it discovered to be down. If this number is not zero, then the longword IP address for each dead (or unreachable) host follows in the output buffer. At most 16 server addresses will be returned, as this is the maximum number of servers for which the *Cache Manager* keeps information.

Among the possible error returns is `ENOENT`, indicating that the optional cell name string input value is not known to the *Cache Manager*.

6.4.5 Cell Operations

The *Cache Manager* is the only active AFS agent that understands the system's cellular architecture. Thus, it keeps important information concerning the identities of the cells in the community, which cell is in direct administrative control of the machine upon which it is running, status and configuration of its own cell, and what cell-specific operations may be legally executed. The following **pioctl()**s allow client processes to access and update this cellular information. Supported operations include adding or updating knowledge of a cell, including the cell overseeing the caller's machine (`VIOCNEWCELL`), fetching the contents of a cell configuration entry (`VIOCGETCELL`), finding out which cell hosts a given file system object (`VIOC_FILE_CELL_NAME`), discovering the cell to which the machine belongs (`VIOC_GET_WS_CELL`), finding out the caller's "primary" cell (`VIOC_GET_PRIMARY_CELL`), and getting/setting certain other per-cell system parameters (`VIOC_GETCELLSTATUS`, `VIOC_SETCELLSTATUS`).

6.4.5.1 `VIOCNEWCELL`: Set cell service information

[Opcode 26] Give the *Cache Manager* all the information it needs to access an AFS cell. Exactly eight longwords are placed at the beginning of the *in* input buffer. These specify the IP addresses for the machine providing AFS authentication and volume location authentication services. The first such longword set to zero will signal the end of the list of server IP addresses. After these addresses, the input buffer hosts the null-terminated name of the cell to which the above servers belong. The *a_pathP* parameter is not used, and so should be set to the null pointer.

Among the possible error returns is `EACCES`, indicating that the caller does not have the necessary rights to perform the operation. Only `root` is allowed to set cell server information. If either the IP address array or the server name is unacceptable, `EINVAL` will be returned.

6.4.5.2 `VIOCGETCELL`: Get cell configuration entry

[Opcode 27] Get the *i*'th cell configuration entry known to the *Cache Manager*. The index of the desired entry is placed into the *in* input buffer as a longword, with the first legal value being zero. If there is a cell associated with the given index, the output buffer will be filled with an array of 8 longwords, followed by a null-terminated string. The

longwords correspond to the list of IP addresses of the machines providing AFS authentication and volume location services. The string reflects the name of the cell for which the given machines are operating. There is no explicit count returned of the number of valid IP addresses in the longword array. Rather, the list is terminated by the first zero value encountered, or when the eighth slot is filled.

This routine is intended to be called repeatedly, with the index starting at zero and increasing each time. The array of cell information records is kept compactly, without holes. A return value of `EDOM` indicates that the given index does not map to a valid entry, and thus may be used as the terminating condition for the iteration.

6.4.5.3 `VIOC_FILE_CELL_NAME`: Get cell hosting a given object

[Opcode 30] Ask the *Cache Manager* to return the name of the cell in which the file system object named by *a_pathP* resides. The input arguments are not used, so *in* should be set to the null pointer and *in_size* should be set to zero. The null-terminated cell name string is returned in the *out* output buffer.

Among the possible error values, `EINVAL` indicates that the pathname provided in *a_pathP* is illegal. If there is no cell information associated with the given object, `ESRCH` is returned.

6.4.5.4 `VIOC_GET_WS_CELL`: Get caller's home cell name

[Opcode 31] Return the name of the cell to which the caller's machine belongs. This cell name is returned as a null-terminated string in the output buffer. The input arguments are not used, so *in* should be set to the null pointer and *in_size* should be set to zero.

Among the possible error returns is `ESRCH`, stating that the caller's home cell information was not available.

6.4.5.5 `VIOC_GET_PRIMARY_CELL`: Get the caller's primary cell

[Opcode 33] Ask the *Cache Manager* to return the name of the caller's primary cell. Internally, the *Cache Manager* scans its user records, and the cell information referenced by that record is used to extract the cell's string name. The input arguments are not used, so *in* should be set to the null pointer and *in_size* should be set to zero. The *a_pathP* pathname argument is not used either, and should similarly be set to the null

pointer. The null-terminated cell name string is placed into the output buffer pointed to by *out* if it has sufficient room.

Among the possible error returns is `ESRCH`, stating that the caller's primary cell information was not available.

6.4.5.6 `VIOC_GETCELLSTATUS`: Get status info for a cell entry

[Opcode 35] Given a cell name, return a single longword of status flags from the *Cache Manager*'s entry for that cell. The null-terminated cell name string is expected to be in the *in* parameter, with *in_size* set to its length plus one for the trailing null. The status flags are returned in the *out* buffer, which must have *out_size* set to `sizeof(long)` or larger.

The *Cache Manager* defines the following output flag values for this operation:

- `0x1` This entry is considered the caller's primary cell.
- `0x2` The UNIX *setuid()* operation is not honored.
- `0x4` An obsolete version of the *Volume Location Server*'s database is being used. While defined, this flag should no longer be set in modern systems.

Among the possible error returns is `ENOENT`, informing the caller that the *Cache Manager* has no knowledge of the given cell name.

6.4.5.7 `VIOC_SETCELLSTATUS`: Set status info for a cell entry

[Opcode 36] Given a cell name and an image of the cell status bits that should be set, record the association in the *Cache Manager*. The input buffer *in* must be set up as follows. The first entry is the longword containing the cell status bits to be set (see the `VIOC_GETCELLSTATUS` description above for valid flag definitions). The next entry is another longword, ignored by the *Cache Manager*. The third and final entry in the input buffer is a null-terminated string containing the name of the cell for which the status flags are to be applied.

Among the possible error returns is `ENOENT`, reflecting the *Cache Manager*'s inability to locate its record for the given cell. Only `root` is allowed to execute this operation, and an `EACCES` return indicates the caller was not effectively `root` when the call took place.

6.4.6 Authentication Operations

The *Cache Manager* serves as the repository for authentication information for AFS clients. Each client process belongs to a single **Process Authentication Group (PAG)**. Each process in a given PAG shares authentication information with the other members, and thus has the identical rights with respect to AFS Access Control Lists (ACLs) as all other processes in the PAG. As the *Cache Manager* interacts with *File Server*s as a client process' agent, it automatically and transparently presents the appropriate authentication information as required in order to gain the access to which the caller is entitled. Each PAG can host exactly one *token* per cell. These tokens are objects that unequivocally codify the principal's identity, and are encrypted for security. Token operations between a *Cache Manager* and *File Server* are also encrypted, as are the interchanges between clients and the *Authentication Server*s that generate these tokens.

There are actually two different flavors of tokens, namely **clear** and **secret**. The data structure representing clear tokens is described in Section 6.2.2, and the secret token appears as an undifferentiated byte stream.

This section describes the operations involving these tokens, namely getting and setting the caller's token for a particular cell (`VIOCGETTOK`, `VIOCSETTOK`), checking a caller's access on a specified file system object (`VIOCACCESS`), checking the status of caller's tokens associated with the set of *File Server* connections maintained on its behalf (`VIOCCKCONN`), and discarding tokens entirely (`VIOCUNLOG`, `VIOCUNPAG`). These abilities are used by such programs as *login*, *klog*, *unlog*, and *tokens*, which must generate, manipulate, and/or destroy AFS tokens.

6.4.6.1 `VIOCSETTOK`: Set the caller's token for a cell

[Opcode 3] Store the caller's secret and clear tokens within the *Cache Manager*. The input buffer is used to hold the following quantities, laid out end to end. The first item placed in the buffer is a longword, specifying the length in bytes of the secret token, followed by the body of the secret token itself. The next field is another longword, this time describing the length in bytes of the `struct ClearToken`, followed by the structure. These are all required fields. The caller may optionally include two additional fields, following directly after the required ones. The first optional field is a longword which is set to a non-zero value if the cell in which these tokens were generated is to be marked as the caller's primary cell. The second optional argument is a null-terminated string specifying the cell in which these tokens apply. If these two optional arguments do not appear, the *Cache Manager* will default to using its home cell and marking the entry as non-primary. The *a_pathP* pathname parameter is not used, and thus should be set to

the null pointer.

If the caller does not have any tokens registered for the cell, the *Cache Manager* will store them. If the caller already has tokens for the cell, the new values will overwrite their old values. Because these are stored per PAG, the new tokens will thus determine the access rights of all other processes belonging to the PAG.

Among the possible error returns is `ESRCH`, indicating the named cell is not recognized, and `EIO`, if information on the local cell is not available.

6.4.6.2 `VIOCGETTOK`: Get the caller's token for a cell

[Opcode 8] Get the specified authentication tokens associated with the caller. The *a_pathP* parameter is not used, so it should be set to the null pointer. Should the input parameter *in* be set to a null pointer, then this call will place the user's tokens for the machine's home cell in the *out* output buffer, if such tokens exist. In this case, the following objects are placed in the output buffer. First, a longword specifying the number of bytes in the body of the secret token is delivered, followed immediately by the secret token itself. Next is a longword indicating the length in bytes of the clear token, followed by the clear token. The input parameter may also consist of a single longword, indicating the index of the token desired. Since the *Cache Manager* is capable of storing multiple tokens per principal, this allows the caller to iteratively extract the full set of tokens stored for the PAG. The first valid index value is zero. The list of tokens is kept compactly, without holes. A return value of `EDOM` indicates that the given index does not map to a valid token entry, and thus may be used as the terminating condition for the iteration.

Other than `EDOM`, another possible error return is `ENOTCONN`, specifying that the caller does not have any AFS tokens whatsoever.

6.4.6.3 `VIOCACCESS`: Check caller's access on object

[Opcode 20] This operation is used to determine whether the caller has specific access rights on a particular file system object. A single longword is placed into the input buffer, *in*, representing the set of rights in question. The acceptable values for these access rights are listen in Section 6.4.5. The object to check is named by the *a_pathP* parameter. The output parameters are not accessed, so *out* should be set to the null pointer, and *out_size* set to zero.

If the call returns successfully, the caller has at least the set of rights denoted by the bits set in the input buffer. Otherwise, `EACCESS` is returned.

6.4.6.4 `VIOCCKCONN`: Check status of caller's tokens/connections

[Opcode 12] Check whether the suite of *File Server* connections maintained on behalf of the caller by the *Cache Manager* has valid authentication tokens. This function always returns successfully, communicating the health of said connections by writing a single longword value to the specified output buffer in *out*. If zero is returned to the output buffer, then two things are true. First, the caller has tokens for at least one cell. Second, all tokens encountered upon a review of the caller's connections have been properly minted (i.e., have not been generated fraudulently), and, in addition, have not yet expired. If these conditions do not currently hold for the caller, then the output buffer value will be set to `EACCES`. Neither the *a_pathP* nor input parameters are used by this call.

6.4.6.5 `VIOCUNLOG`: Discard authentication information

[Opcode 9] Discard all authentication information held in trust for the caller. The *Cache Manager* sweeps through its user records, destroying all of the caller's associated token information. This results in reducing the rights of all processes within the caller's PAG to the level of file system access granted to the special `system:anyuser` group.

This operation always returns successfully. None of the parameters are referenced, so they should all be set to null pointers and zeroes as appropriate.

6.4.6.6 `VIOCUNPAG`: Discard authentication information

[Opcode 21] This call is essentially identical to the `VIOCUNLOG` operation, and is in fact implemented internally by the same code for `VIOCUNLOG`.

6.4.7 ACL Operations

This set of opcodes allows manipulation of AFS Access Control Lists (ACLs). Callers are allowed to fetch the ACL on a given directory, or to set the ACL on a directory. In AFS-3, ACLs are only maintained on directories, not on individual files. Thus, a directory ACL determines the allowable accesses on all objects within that directory in conjunction with their normal UNIX mode (owner) bits. Should the *a_pathP* parameter specify a file instead of a directory, the ACL operation will be performed on the directory in which the given file resides.

These **pioctl()** opcodes deal only in *external formats* for ACLs, namely the actual text stored in an AFS ACL container. This external format is a character string, composed of a descriptive header followed by some number of individual principal-rights pairs. AFS ACLs actually specify *two* sublists, namely the *positive* and *negative rights* lists. The positive list catalogues the set of rights that certain principals (individual users or groups of users) have, while the negative list contains the set of rights specifically denied to the named parties.

These external ACL representations differ from the *internal format* generated by the *Cache Manager* after a parsing pass. The external format may be easily generated from the internal format as follows. The header format is expressed with the following *printf()* statement:

```
printf("%d\n%d\n", NumPositiveEntries, NumNegativeEntries);
```

The header first specifies the number of entries on the positive rights list, which appear first in the ACL body. The number of entries on the negative list is the second item in the header. The negative entries appear after the last positive entry.

Each entry in the ACL proper obeys the format imposed by the following *printf()* statement:

```
printf("%s\t%d\n", UserOrGroupName, RightsMask);
```

Note that the string name for the user or group is stored in an externalized ACL entry. The *Protection Server* stores the mappings between the numerical identifiers for AFS principals and their character string representations. There are cases where there is no mapping from the numerical identifier to a string name. For example, a user or group may have been deleted sometime after they were added to the ACL and before the *Cache Manager* externalized the ACL for storage. In this case, the *Cache Manager* sets *UserOrGroupName* to the string version of the principal's integer identifier. Should the `erz` principal be deleted from the *Protection Server*'s database in the above scenario, then the string "`1019`" will be stored, since it corresponded to `erz`'s former numerical identifier.

The *RightsMask* parameter to the above call represents the set of rights the named principal may exercise on the objects covered by the ACL. The following flags may be OR'ed together to construct the desired access rights placed in *RightsMask*:

```
#define PRSFS_READ          1 /*Read files*/
```

```
#define PRSFS_WRITE        2 /*Write & write-lock existing files*/
#define PRSFS_INSERT       4 /*Insert & write-lock new files*/
#define PRSFS_LOOKUP       8 /*Enumerate files and examine ACL*/
#define PRSFS_DELETE      16 /*Remove files*/
#define PRSFS_LOCK        32 /*Read-lock files*/
#define PRSFS_ADMINISTER  64 /*Set access list of directory*/
```

6.4.7.1 VIOCSETAL: Set the ACL on a directory

[Opcode 1] Set the contents of the ACL associated with the file system object named by *a_pathP*. Should this pathname indicate a file and not a directory, the *Cache Manager* will apply this operation to the file's parent directory. The new ACL contents, expressed in their externalized form, are made available in *in*, with *in_size* set to its length in characters, including the trailing null. There is no output from this call, so *out_size* should be set to zero. Internally, the *Cache Manager* will call the *RXAFS_StoreACL()* RPC (see Section 5.1.3.3 to store the new ACL on the proper *File Server*.

Possible error codes include `EINVAL`, indicating that one of three things may be true: the named path is not in AFS, there are too many entries in the specified ACL, or a non-existent user or group appears on the ACL.

6.4.7.2 VIOCGETAL: Get the ACL for a directory

[Opcode 2] Get the contents of the ACL associated with the file system object named by *a_pathP*. Should this pathname indicate a file and not a directory, the *Cache Manager* will apply this operation to the file's parent directory. The ACL contents, expressed in their externalized form, are delivered into the *out* buffer if *out_size* has been set to a value which indicates that there is enough room for the specified ACL. This ACL string will be null-terminated. There is no input to this call, so *in_size* should be set to zero. Internally, the *Cache Manager* will call the *RXAFS_FetchACL()* RPC (see Section 5.1.3.1) to fetch the ACL from the proper *File Server*.

Possible error codes include `EINVAL`, indicating that the named path is not in AFS.

6.4.8 Cache Operations

It is possible to inquire about and affect various aspects of the cache maintained locally by the *Cache Manager* through the group of **pioctl()**s described below. Specifically, one may force certain file system objects to be removed from the cache (`VIOCFLUSH`), set the maximum number of blocks usable by the cache (`VIOCSETCACHESIZE`), and ask for information about the cache's current state (`VIOCGETCACHEPARAMS`).

6.4.8.1 `VIOCFLUSH`: Flush an object from the cache

[Opcode 6] Flush the file system object specified by *a_pathP* out of the local cache. The other parameters are not referenced, so they should be set to the proper combination of null pointers and zeroes.

Among the possible error returns is `EINVAL`, indicating that the value supplied in the *a_pathP* parameter is not acceptable.

6.4.8.2 `VIOCSETCACHESIZE`: Set maximum cache size in blocks

[Opcode 24] Instructs the *Cache Manager* to set a new maximum size (in 1 Kbyte blocks) for its local cache. The input buffer located at *in* contains the new maximum block count. If zero is supplied for this value, the *Cache Manager* will revert its cache limit to its value at startup time. Neither the *a_pathP* nor output buffer parameters is referenced by this operation. The *Cache Manager* recomputes its other cache parameters based on this new value, including the number of cache files allowed to be dirty at once and the total amount of space filled with dirty chunks. Should the new setting be smaller than the number of blocks currently being used, the *Cache Manager* will throw things out of the cache until it obeys the new limit.

The caller is required to be effectively running as `root`, or this call will fail, returning `EACCES`. If the *Cache Manager* is configured to run with a memory cache instead of a disk cache, this operation will also fail, returning `EROF`.

6.4.8.3 `VIOCGETCACHEPARAMS`: Get current cache parameter values

[Opcode 40] Fetch the current values being used for the cache parameters. The output buffer is filled with `MAXGCSTATS` (16) longwords, describing these parameters. Only the first two longwords in this array are currently set. The first contains the value of `afs_cacheBlocks`, or the maximum number of 1 Kbyte blocks which may be used in the cache (see Section 6.4.8.2 for how this value may be set). The second longword contains the value of the *Cache Manager*'s internal `afs_blocksUsed` variable, or the number of these cache blocks currently in use. All other longwords in the array are set to zero. Neither the *a_pathP* nor input buffer arguments are referenced by this call.

This routine always returns successfully.

6.4.9 Miscellaneous Operations

There are several other AFS-specific operations accessible via the **pioctl()** interface that don't fit cleanly into the above categories. They are described in this section, and include manipulation of the socket-based Mariner file trace interface (`VIOC_AFS_MARINER_HOST`), enabling and disabling of the file-based *AFSLog* output interface for debugging (`VIOC_VENUSLOG`), getting and setting the value of the special `@sys` pathname component mapping (`VIOC_AFS_SYSNAME`), and turning the NFS-AFS translator service on and off (`VIOC_EXPORTAFS`).

6.4.9.1 `VIOC_AFS_MARINER_HOST`: Get/set file transfer monitoring output

[Opcode 32] This operation is used to get or set the IP address of the host destined to receive Mariner output. A detailed description of the *Cache Manager* Mariner interface may be found in Section 6.7.

The input buffer located at *in* is used to pass a single longword containing the IP address of the machine to receive output regarding file transfers between the *Cache Manager* and any *File Server*. If the chosen host IP address is `0xffffffff`, the *Cache Manager* is prompted to turn off generation of Mariner output entirely. If the chosen host IP address is zero, then the *Cache Manager* will not set the Mariner host, but rather return the current Mariner host as a single longword written to the *out* output buffer. Any other value chosen for the host IP address enables Mariner output (if it was not already enabled) and causes all further traffic to be directed to the given machine.

This function always returns successfully.

6.4.9.2 `VIOC_VENUSLOG`: Enable/disable *Cache Manager* logging

[Opcode 34] Tell the *Cache Manager* whether to generate debugging information, and what kind of debugging output to enable. The input buffer located at *in* is used to transmit a single longword to the *Cache Manager*, expressing the caller's wishes. Of the four bytes making up the longword, the highest byte indicates the desired value for the internal `afsDebug` variable, enabling or disabling general trace output. The next highest byte indicates the desired value for the internal `netDebug` variable, enabling or disabling network-level debugging traces. The third byte is unused, and the low-order byte

represents an overall on/off value for the functionality. There is a special value for the low-order byte, `99`, which instructs the *Cache Manager* to return the current debugging setting as a single longword placed into the output buffer pointed to by *out*. The *a_pathP* parameter is not referenced by this routine.

Trace output is delivered to the *AFSLog* file, typically located in the */usr/vice/etc* directory. When this form of debugging output is enabled, the existing *AFSLog* file is truncated, and its file descriptor is stored for future use. When this debugging is disabled, a *close()* is done on the file, forcing all its data to disk. For additional information on the *AFSLog* file for collecting *Cache Manager* traces, please see the description in Section 6.6.2.1.

This call will only succeed if the caller is effectively running as `root`. If this is not the case, an error code of `EACCES` is returned.

6.4.9.3 `VIOC_AFS_SYSNAME`: Get/set the `@sys` mapping

[Opcode 38] Get or set the value of the special *@sys* pathname component understood by the *Cache Manager*. The input buffer pointed to by *in* is used to house a longword whose value determines whether the `@sys` value is being set (1) or whether the current value is being fetched (0). If it is being set, then a null-terminated string is expected to follow in the input buffer, specifying the new value of `@sys`. Otherwise, if we are asking the *Cache Manager* for the current `@sys` setting, a null-terminated string bearing that value will be placed in the `out` output buffer. The *a_pathP* parameter is not used by this call, and thus should be set to a null pointer.

There are no special privileges required of the caller to fetch the value of the current `@sys` mapping. However, a native caller must be running effectively as `root` in order to successfully alter the mapping. An unauthorized attempt to change the `@sys` setting will be ignored, and cause this routine to return `EACCES`. This requirement is relaxed for `VIOC_AFS_SYSNAME` **pioctl()** calls emanating from foreign file systems such as NFS and accessing AFS files through the NFS-AFS translator. Each such remote caller may set its own notion of what the `@sys` mapping is without affecting native AFS clients. Since the uid values received in calls from NFS machines are inherently insecure, it is impossible to enforce the fact that the caller is truly `root` on the NFS machine. This, while any principal running on an NFS machine may change that foreign machine's perception of `@sys`, it does not impact native AFS users in any way.

6.4.9.4 `VIOC_EXPORTAFS`: Enable/disable NFS/AFS translation

[Opcode 39] Enable or disable the ability of an AFS-capable machine to export AFS access to NFS clients. Actually, this is a general facility allowing exportation of AFS service to any number of other file systems, but the only support currently in place is for NFS client machines. A single longword is expected in the input buffer *in*. This input longword is partitioned into individual bytes, organized as follows. The high-order byte communicates the type of foreign client to receive AFS file services. There are currently two legal values for this field, namely 0 for the null foreign file system and 1 for NFS. The next byte determines whether the *Cache Manager* is being asked to get or set this information. A non-zero value here is interpreted as a command to set the export information according to what's in the input longword, and a zero-valued byte in this position instructs the *Cache Manager* to place a longword in the output buffer *out*, which contains the current export settings for the foreign system type specified in the high-order byte. The third input byte is not used, and the lowest-order input buffer byte determines whether export services for the specified system are being enabled or disabled. A non-zero value will turn on the services, and a zero value will shut them down. The *a_pathP* pathname parameter is not used by this call, and the routine generates output only if the export information is being requested instead of being set.

The caller must be effectively running as `root` in order for this operation to succeed. The call returns `EACCES` if the caller is not so authorized. If the caller specifies an illegal foreign system type in the high-order byte of the input longword, then `ENODEV` is returned. Again, NFS is the only foreign file system currently supported.

Practically speaking, the machine providing NFS-AFS translation services must enable this service with this **pioctl()** before any NFS client machines may begin accessing AFS files. Conversely, if an administrator turns off this export facility, the export code on the translator machine will immediately stop responding to traffic from its active NFS clients.

6.5 RPC Interface

6.5.1 Introduction

This section covers the structure and workings of the *Cache Manager*'s RPC interface. Typically, these calls are made by *File Server* processes. However, some of the calls are designed specifically for debugging programs (e.g., the `cmdebug` facility) and for collection of statistical and performance information from the *Cache Manager*. Any client application that makes direct calls on the *File Server* RPC interface must be prepared to export a subset of the *Cache Manager* RPC interface, as discussed in Section 5.1.6.

This section will first examine the *Cache Manager*'s use of locks, whose settings may be observed via one of the RPC interface calls. Next, it will present some definitions and data structures used in the RPC interface, and finally document the individual calls available through this interface.

6.5.2 Locks

The *Cache Manager* makes use of locking to insure its internal integrity in the face of its multi-threaded design. A total of 11 locks are maintained for this purpose, one of which is now obsolete and no longer used (see below). These locks are strictly internal, and the *Cache Manager* itself is the only one able to manipulate them. The current settings for these system locks are externally accessible for debugging purposes via the *AFSRXCB_GetLock()* RPC interface call, as described in Section 6.5.5.4. For each lock, its index in the locking table is given in the following text.

- **afs_xvcache** *[Index 0]*: This lock controls access to the status cache entries maintained by the *Cache Manager*. This stat cache keeps *stat()*-related information for AFS files it has dealt with. The stat information is kept separate from actual data contents of the related file, since this information may change independently (say, as a result of a UNIX *chown() call.*
- **afs_xdcache** *[Index 1]*: This lock moderates access to the *Cache Manager*'s data cache, namely the contents of the file system objects it has cached locally. As stated above, this data cache is separate from the associated *stat()* information.
- **afs_xserver** *[Index 2]*: This lock controls access to the *File Server* machine description table, which keeps tabs on all *File Server*s contacted in recent history. This lock thus indirectly controls access to the set of per-server RPC connection descriptors the *File Server* table makes visible.
- **afs_xvcb** *[Index 3]*: This lock supervises access to the volume callback information kept by the *Cache Manager*. This table is referenced, for example, when a client decides to remove one or more callbacks on files from a given volume (see the *RXAFS_GiveUpCallBacks()* description on Section 5.1.3.13).
- **afs_xbrs** *[Index 4]*: This lock serializes the actions of the *Cache Manager*'s background daemons, which perform prefetching and background file storage duties.
- **afs_xcell** *[Index 5]*: This lock controls the addition, deletion, and update of items on the linked list housing information on cells known to the *Cache Manager*.

- **afs_xconn** *[Index 6]*: This lock supervises operations concerning the set of RPC connection structures kept by the system. This lock is used in combination with the `afs_xserver` lock described above. In some internal *Cache Manager* code paths, the *File Server* description records are first locked, and then the `afs_xconn` lock is used to access the associated *Rx* connection records.

- **afs_xuser** *[Index 7]*: This lock serializes access to the per-user structures maintained by the *Cache Manager*.

- **afs_xvolume** *[Index 8]*: This lock is used to control access to the *Cache Manager*'s volume information cache, namely the set of entries currently in memory, a subset of those stably housed in the *VolumeItems* disk file (see Section 6.6.2.3).

- **afs_puttofileLock** *[Index 9]*: This lock is obsolete, and while still defined by the system is no longer used. It formerly serialized writes to a debugging output interface buffer, but the internal mechanism has since been updated and improved.

- **afs_ftf** *[Index 10]*: This lock is used when flushing cache text pages from the machine's virtual memory tables. For each specific machine architecture on which the *Cache Manager* runs, there is a set of virtual memory operations which must be invoked to perform this operation. The result of such activities is to make sure that the latest contents of new incarnations of binaries are used, instead of outdated copies of previous versions still resident in the virtual memory system.

6.5.3 Definitions and Typedefs

This section documents some macro definitions and typedefs referenced by the *Cache Manager*'s RPC interface. Specifically, these definitions and typedefs are used in the *RXAFSCB_GetXStats()* and *RXAFSCB_XStatsVersion* calls as described in Sections 6.5.5.6 and 6.5.5.7.

```
/*
 * Define the version of CacheManager and FileServer extended
 * statistics being implemented.
 */
const AFSCB_XSTAT_VERSION = 2;

/*
 * Define the maximum arrays for passing extended statistics
 * info for the CacheManager and FileServer back to our caller.
 */
const AFSCB_MAX_XSTAT_LONGS = 2048;
```

```
typedef long AFSCB_CollData<AFSCB_MAX_XSTAT_LONGS>;

/*
 * Define the identifiers for the accessible extended stats data
 * collections.
 */
const AFSCB_XSTATSCOLL_CALL_INFO = 0;         /*CM call counting & info*/
const AFSCB_XSTATSCOLL_PERF_INFO = 1;         /*CM performance info*/
const AFSCB_XSTATSCOLL_FULL_PERF_INFO = 2;    /*CM extended performance info*/
```

6.5.4 Structures

This section documents some structures used in the *Cache Manager* RPC interface. As with the constants and typedefs in the previous section, these items are used in the *RXAFSCB_GetXStats()* and *RXAFSCB_XStatsVersion* calls as described in Sections 6.5.5.6 and 6.5.5.7.

6.5.4.1 struct afs_MeanStats

This structure may be used to collect a running average figure. It is included in some of the statistics structures described below.

Fields

`long average` - The computed average.

`long elements` - The number of elements sampled for the above average.

6.5.4.2 struct afs_CMCallStats

This structure maintains profiling information, communicating the number of times internal *Cache Manager* functions are invoked. Each field name has a "`C_`" prefix, followed by the name of the function being watched. Because this structure has entries for over 500 functions, it will not be described further here. For the full layout of this structure, see Appendix 7.

The `AFSCB_XSTATSCOLL_CALL_INFO` data collection includes the information in this structure.

6.5.4.3 struct afs_CMMeanStats

This is the other part of the information (along with the `struct afs_CMCallStats` construct described above) returned by the `AFSCB_XSTATSCOLL_CALL_INFO` data collection defined by the *Cache Manager* (see Section 6.5.3). It is accessible via the *RXAFSCB_GetXStats()* interface routine, as defined in Section 6.5.5.7.

This structure represents the beginning of work to compute average values for some of the extended statistics collected by the *Cache Manager*.

Fields

`struct afs_MeanStats something` - Intended to collect averages for some of the *Cache Manager* extended statistics; not yet implemented.

6.5.4.4 struct afs_CMStats

This structure defines the information returned by the `AFSCB_XSTATSCOLL_CALL_INFO` data collection defined by the *Cache Manager* (see Section 6.5.3). It is accessible via the *RXAFSCB_GetXStats()* interface routine, as defined in Section 6.5.5.7.

Fields

`struct afs_CMCallStats callInfo` - Contains the counts on the number of times each internal *Cache Manager* function has been called.

`struct afs_CMMeanStats meanInfo` - Intended to collect averages for some of the *Cache Manager* extended statistics; not yet implemented.

6.5.4.5 struct afs_stats_SrvUpDownInfo

This structure is used to track aggregate up/downtime statistics for file servers and volume location servers.

Fields

`long numTtlRecords` - Number of records, active or inactive.

`long numUpRecords` - Number of active records currently marked up.

`long numDownRecords` - Number of active records currently marked down.

`long sumOfRecordAges` - Sum of server record lifetimes.

`long ageOfYoungestRecord` - Age of youngest server record.

`long ageOfOldestRecord` - Age of oldest server record.

`long numDowntimeIncidents` - Number of completed downtime incidents.

`long numRecordsNeverDown` - Number of server records never marked down.

`long maxDowntimesInARecord` - Longest downtime seen by any record.

`long sumOfDowntimes` - Sum of all completed downtimes, in seconds.

`long shortestDowntime` - Shortest downtime, in seconds.

`long longestDowntime` - Longest downtime, in seconds.

`long downDurations[AFS_STATS_NUM_DOWNTIME_DURATION_BUCKETS]` - Statistics on downtime durations.

`long downIncidents[AFS_STATS_NUM_DOWNTIME_INCIDENTS_BUCKETS]` - Statistics on number of downtime incidents.

The array `downDurations` counts downtimes of certain length. Its indices are related to the following distribution:

```
/*
 * Constants to track downtime durations:
 *      Bucket 0:           dur <= 10 min
 *      Bucket 1: 10 min  < dur <= 30 min
 *      Bucket 2: 30 min  < dur <= 1 hour
 *      Bucket 3: 1 hour  < dur <= 2 hours
 *      Bucket 4: 2 hours < dur <= 4 hours
 *      Bucket 5: 4 hours < dur <= 8 hours
 *      Bucket 6:           dur >= 8 hours
 */
#define AFS_STATS_NUM_DOWNTIME_DURATION_BUCKETS       7

#define AFS_STATS_MAX_DOWNTIME_DURATION_BUCKET0     600 /*10 minutes*/
```

```
#define AFS_STATS_MAX_DOWNTIME_DURATION_BUCKET1    1800 /*30 minutes*/
#define AFS_STATS_MAX_DOWNTIME_DURATION_BUCKET2    3600 /*60 minutes*/
#define AFS_STATS_MAX_DOWNTIME_DURATION_BUCKET3    7200 /*2 hours*/
#define AFS_STATS_MAX_DOWNTIME_DURATION_BUCKET4   14400 /*4 hours*/
#define AFS_STATS_MAX_DOWNTIME_DURATION_BUCKET5   28800 /*8 hours*/
```

The array `downIncidents` counts servers with certain number of downtimes. Its indices are related to the following distribution:

```
/*
 * Constants to track downtime incidents:
 *      Bucket 0:           down  =  0 times
 *      Bucket 1:           down  =  1 time
 *      Bucket 2:  1 time  < down <=  5 times
 *      Bucket 3:  5 times < down <= 10 times
 *      Bucket 4: 10 times < down <= 50 times
 *      Bucket 5:           down >  50 times
 */
#define AFS_STATS_NUM_DOWNTIME_INCIDENTS_BUCKETS 6

#define AFS_STATS_MAX_DOWNTIME_INCIDENTS_BUCKET0   0
#define AFS_STATS_MAX_DOWNTIME_INCIDENTS_BUCKET1   1
#define AFS_STATS_MAX_DOWNTIME_INCIDENTS_BUCKET2   5
#define AFS_STATS_MAX_DOWNTIME_INCIDENTS_BUCKET3   10
#define AFS_STATS_MAX_DOWNTIME_INCIDENTS_BUCKET4   50
```

6.5.4.6 struct afs_stats_CMPerf

This is the information returned by the `AFSCB_XSTATSCOLL_PERF_INFO` data collection defined by the *Cache Manager* (see Section 6.5.3). It is accessible via the *RXAFSCB_GetXStats()* interface routine, as defined in Section 6.5.5.7.

Fields

`long numPerfCalls` - Number of performance calls received.

`long epoch` - *Cache Manager* epoch time.

`long numCellsVisible` - Number of cells it knows about.

`long numCellsContacted` - Number of cells contacted.

`long dlocalAccesses` - Number of data accesses to files within the local cell.

`long vlocalAccesses` - Number of stat accesses to files within the local cell.

`long dremoteAccesses` - Number of data accesses to files outside of the local cell.

`long vremoteAccesses` - Number of stat accesses to files outside of the local cell.

`long cacheNumEntries` - Number of cache entries.

`long cacheBlocksTotal` - Number of (1K) blocks configured for the AFS cache.

`long cacheBlocksInUse` - Number of cache blocks actively in use.

`long cacheBlocksOrig` - Number of cache blocks configured at bootup.

`long cacheMaxDirtyChunks` - Maximum number of dirty cache chunks tolerated.

`long cacheCurrDirtyChunks` - Current count of dirty cache chunks.

`long dcacheHits` - Number of data file requests satisfied by the local cache.

`long vcacheHits` - Number of stat entry requests satisfied by the local cache.

`long dcacheMisses` - Number of data file requests *not* satisfied by the local cache.

`long vcacheMisses` - Number of stat entry requests *not* satisfied by the local cache.

`long cacheFlushes` - Number of files flushed from the cache.

`long cacheFilesReused` - Number of cache files reused.

`long ProtServerAddr` - IP addres of the *Protection Server* used *(not implemented)*.

`long vcacheXAllocs` - Additionally allocated vcaches.

`long dcacheXAllocs` - Additionally allocated dcaches.

`long bufAlloced` - Number of buffers allocated by afs.

`long bufHits` - Number of pages found on buffer cache.

`long bufMisses` - Number of pages not found on buffer cache.

`long bufFlushDirty` - Number of cached dirty buffers flushed because all were busy.

`long LargeBlocksActive` - Number of currently used large free pool entries.

`long LargeBlocksAlloced` - Number of allocated large free pool entries.

`long SmallBlocksActive` - Number of currently used small free pool entries.

`long SmallBlocksAlloced` - Number of allocated used small free pool entries.

`long OutStandingMemUsage` - Size of allocated memory.

`long OutStandingAllocs` - Outstanding osi_allocs (no osi_frees yet).

`long CallBackAlloced` - Number callback structures allocated.

`long CallBackFlushes` - Numer callback flush operations performed.

`long srvRecords` - Number of servers currently on record.

`long srvRecordsHWM` - Server record high water mark.

`long srvNumBuckets` - Number of server hash chain buckets.

`long srvMaxChainLength` - Longest server hash chain length.

`long srvMaxChainLengthHWM` - Server hash chain high water mark.

`long sysName_ID` - Sysname ID for host hardware.

`struct afs_stats_SrvUpDownInfo fs_UpDown[2]` - File server downtimes.

`struct afs_stats_SrvUpDownInfo vl_UpDown[2]` - Volume location server downtimes.

`long spare[16]` - Spares for future expansion.

Indices for the `fs_UpDown` and the `vl_UpDown` array are defined by constants `define AFS_STATS_UPDOWN_IDX_SAME_CELL` and `define AFS_STATS_UPDOWN_IDX_DIFF_CELL` with values 0 and 1, respectively.

6.5.4.7 struct afs_stats_opTimingData

This is the structure that maintains detailed statistics on *File Server* or *Cache Manager* calls.

Fields

`long numOps` - Number of operations executed.

`long numSuccesses` - Number of successful operations.

`struct timeval sumTime` - Sum of sample timings.

`struct timeval sqrTime` - Sum of squares of sample timings.

`struct timeval minTime` - Minimum timing value observed.

`struct timeval maxTime` - Maximum timing value observed.

6.5.4.8 struct afs_stats_xferData

This is the structure that maintains detailed statistics regarding *File Server* data transfer operations.

Fields

`long numXfers` - Number of transfers.

`long numSuccesses` - Number of successful transfers.

`struct timeval sumTime` - Sum of sample timings.

`struct timeval sqrTime` - Sum of squares of sample timings.

`struct timeval minTime` - Minimum sample time recorded.

`struct timeval maxTime` - Maximum sample time recorded.

`long sumBytes` - Sum of bytes transferred (in KB).

`long minBytes` - Minimum value observed.

`long maxBytes` - Maximum value observed.

`long count[AFS_STATS_NUM_XFER_BUCKETS]` - Tally for each range of bytes.

The value of `AFS_STATS_NUM_XFER_BUCKETS` constant is 9. The array `count` tallies each transfer according to the number of bytes transferred: 0-128B, 129-1024B, etc. The following constants identify the ranges:

```
#define AFS_STATS_MAXBYTES_BUCKET0     128
#define AFS_STATS_MAXBYTES_BUCKET1    1024
#define AFS_STATS_MAXBYTES_BUCKET2    8192
#define AFS_STATS_MAXBYTES_BUCKET3   16384
#define AFS_STATS_MAXBYTES_BUCKET4   32768
#define AFS_STATS_MAXBYTES_BUCKET5  131072
#define AFS_STATS_MAXBYTES_BUCKET6  524288
#define AFS_STATS_MAXBYTES_BUCKET7 1048576
```

For example `count[0]` records number of transfers with 0-128B of data, `count[1]` records number of transfers with 129-1024B of data, and `count[9]` records number of transfers with more than 1,048,576B of data.

Note: Both the `afs_stats_opTimingData` and `afs_stats_xferData` structures contain a field `sqrTime`. This field represents a sum of squares of timings (the simple sum is kept in the `sumTime`) and can be used for computing standard deviation of timings.

6.5.4.9 struct afs_stats_RPCErrors

This is the structure that maintains records of RPC error analysis outcomes.

Fields

`long err_Server` - Server down error.

`long err_Network` - Network error.

`long err_Protection` - Protection violation.

`long err_Volume` - Volume related error.

`long err_VolumeBusies` - Volume busy conditions encountered.

`long err_Other` - Miscellaneous errors.

6.5.4.10 struct afs_stats_RPCOpInfo

This structure encapsulates information related to performance of RPC calls and transfers of data.

Fields

`struct afs_stats_opTimingData fsRPCTimes[AFS_STATS_NUM_FS_RPC_OPS]` - Individual *File Server* RPC call timings.

`struct afs_stats_RPCErrors fsRPCErrors[AFS_STATS_NUM_FS_RPC_OPS]` - Individual *File Server* RPC call errors.

`struct afs_stats_xferData fsXferTimes[AFS_STATS_NUM_FS_XFER_OPS]` - Individual *File Server* RPC data transfer statistics.

`struct afs_stats_opTimingData cmRPCTimes[AFS_STATS_NUM_CM_RPC_OPS]` - Individual *Cache Manager* RPC call timings.

The value of the `AFS_STATS_NUM_FS_RPC_OPS` constant is 28 and indices of the `fsRPCTimes` and the `fsRPCErrors` array correspond to the following constants and related *FileServer* RPC calls:

```
#define AFS_STATS_FS_RPCIDX_FETCHDATA          0
#define AFS_STATS_FS_RPCIDX_FETCHACL           1
#define AFS_STATS_FS_RPCIDX_FETCHSTATUS        2
#define AFS_STATS_FS_RPCIDX_STOREDATA          3
#define AFS_STATS_FS_RPCIDX_STOREACL           4
#define AFS_STATS_FS_RPCIDX_STORESTATUS        5
#define AFS_STATS_FS_RPCIDX_REMOVEFILE         6
```

```
#define AFS_STATS_FS_RPCIDX_CREATEFILE          7
#define AFS_STATS_FS_RPCIDX_RENAME              8
#define AFS_STATS_FS_RPCIDX_SYMLINK             9
#define AFS_STATS_FS_RPCIDX_LINK               10
#define AFS_STATS_FS_RPCIDX_MAKEDIR            11
#define AFS_STATS_FS_RPCIDX_REMOVEDIR          12
#define AFS_STATS_FS_RPCIDX_SETLOCK            13
#define AFS_STATS_FS_RPCIDX_EXTENDLOCK         14
#define AFS_STATS_FS_RPCIDX_RELEASELOCK        15
#define AFS_STATS_FS_RPCIDX_GETSTATISTICS      16
#define AFS_STATS_FS_RPCIDX_GIVEUPCALLBACKS    17
#define AFS_STATS_FS_RPCIDX_GETVOLUMEINFO      18
#define AFS_STATS_FS_RPCIDX_GETVOLUMESTATUS    19
#define AFS_STATS_FS_RPCIDX_SETVOLUMESTATUS    20
#define AFS_STATS_FS_RPCIDX_GETROOTVOLUME      21
#define AFS_STATS_FS_RPCIDX_CHECKTOKEN         22
#define AFS_STATS_FS_RPCIDX_GETTIME            23
#define AFS_STATS_FS_RPCIDX_NGETVOLUMEINFO     24
#define AFS_STATS_FS_RPCIDX_BULKSTATUS         25
#define AFS_STATS_FS_RPCIDX_XSTATSVERSION      26
#define AFS_STATS_FS_RPCIDX_GETXSTATS          27
```

The value of the `AFS_STATS_NUM_FS_XFER_OPS` constant is 2 and indices of the `fsXferTimes` array correspond to the following constants:

```
#define AFS_STATS_FS_XFERIDX_FETCHDATA          0
#define AFS_STATS_FS_XFERIDX_STOREDATA          1
```

The value of the `AFS_STATS_NUM_CM_RPC_OPS` constant is 7 and indices of the `cmRPCTimes` array correspond to the following constants and related RPC calls:

```
#define AFS_STATS_CM_RPCIDX_CALLBACK            0
#define AFS_STATS_CM_RPCIDX_INITCALLBACKSTATE   1
#define AFS_STATS_CM_RPCIDX_PROBE               2
#define AFS_STATS_CM_RPCIDX_GETLOCK             3
#define AFS_STATS_CM_RPCIDX_GETCE               4
#define AFS_STATS_CM_RPCIDX_XSTATSVERSION       5
#define AFS_STATS_CM_RPCIDX_GETXSTATS           6
```

6.5.4.11 struct afs_stats_AuthentInfo

This structure contains statistics regarding *Cache Manager* authentication information.

Fields

`long curr_PAGs` - Current number of PAGs.

`long curr_Records` - Current number of records in the table.

`long curr_AuthRecords` - Current number of authenticated records (with valid ticket).

`long curr_UnauthRecords` - Current number of unauthenticated records (without any ticket at all).

`long curr_MaxRecordsInPAG` - Maximum records for a single PAG.

`long curr_LongestChain` - Length of longest current hash chain.

`long PAGCreations` - Number of PAG creations.

`long TicketUpdates` - Number of ticket additions and refreshes.

`long HWM_PAGs` - High water mark for the number of PAGs.

`long HWM_Records` - High water mark for the number of records.

`long HWM_MaxRecordsInPAG` - High water mark for a maximum records in a single PAG.

`long HWM_LongestChain` - High water mark for the longest hash chain.

6.5.4.12 struct afs_stats_AccessInfo

This structure contains statistics regarding replicated and unreplicated file access.

Fields

`long unreplicatedRefs` - Number of references to unreplicated data.

`long replicatedRefs` - Number of references to replicated data.

`long numReplicasAccessed` - Number of replicas accessed.

`long maxReplicasPerRef` - Maximum number of replicas accessed per reference.

`long refFirstReplicaOK` - Number of references satisfied by the first replica accessed.

6.5.4.13 struct afs_CMPerfStats

This is the information returned by the `AFSCB_XSTATSCOLL_FULL_PERF_INFO` data collection defined by the *Cache Manager* (see Section 6.5.3). It is accessible via the *RXAFSCB_GetXStats()* interface routine, as defined in Section 6.5.5.7.

Fields

`long numFullPerfCalls` - Number of accesses.

`struct afs_stats_CMPerf perf` - General performance statistics.

`struct afs_stats_RPCOpInfo rpc` - RPC call statistics.

`struct afs_stats_AuthentInfo authent` - Authentication statistics.

`struct afs_stats_AccessInfo accessinf` - Access statistics.

`struct afs_stats_AuthorInfo author` - Authorship statistics (not implemented).

6.5.5 Function Calls

This section discusses the *Cache Manager* interface calls. No special permissions are required of the caller for any of these operations. A summary of the calls making up the interface appears below:

- *RXAFSCB_Probe()* "Are-you-alive" call.
- *RXAFSCB_CallBack()* Report callbacks dropped by a *File Server*.
- *RXAFSCB_InitCallBackState()* Purge callback state from a *File Server*.
- *RXAFSCB_GetLock()* Get contents of *Cache Manager* lock table.
- *RXAFSCB_GetCE()* Get cache file description.
- *RXAFSCB_XStatsVersion()* Get version of extended statistics package.
- *RXAFSCB_GetXStats()* Get contents of extended statistics data collection.

6.5.5.1 RXAFSCB_Probe — Acknowledge that the underlying callback service is still operational

`int` **RXAFSCB_Probe**`(IN struct rx_call *a_rxCallP)`

Description

[Opcode 206] This call simply implements an "are-you-alive" operation, used to determine if the given *Cache Manager* is still running. Any *File Server* will probe each of the *Cache Manager*s with which it has interacted on a regular basis, keeping track of their health. This information serves an important purpose for a *File Server*. In particular, it is used to trigger purging of deceased *Cache Manager*s from the *File Server*'s callback records, and also to instruct a new or "resurrected" *Cache Manager* to purge its own callback state for the invoking *File Server*.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`---` No error codes are generated.

6.5.5.2 RXAFSCB_CallBack — Report callbacks dropped by a *File Server*

```
int RXAFSCB_CallBack(IN struct rx_call *a_rxCallP,
                     IN AFSCBFids *a_fidArrayP,
                     IN AFSCBs *a_callBackArrayP)
```

Description

[Opcode 204] Provide information on dropped callbacks to the *Cache Manager* for the calling *File Server*. The number of fids involved appears in *a_fidArrayP->AFSCBFids_len*, with the fids themselves located at *a_fidArrayP->AFSCBFids_val*. Similarly, the number of associated callbacks is placed in *a_callBackArrayP->AFSCBs_len*, with the callbacks themselves located at *a_callBackArrayP->AFSCBs_val*.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`---` No error codes are generated.

6.5.5.3 RXAFSCB_InitCallBackState — Purge callback state from a *File Server*

`int` **RXAFSCB_InitCallBackState**`(IN struct rx_call *a_rxCallP)`

Description

[Opcode 205] This routine instructs the *Cache Manager* to purge its callback state for all files and directories that live on the calling host. This function is typically called by a *File Server* when it gets a request from a *Cache Manager* that does not appear in its internal records. This handles situations where *Cache Manager*s survive a *File Server*, or get separated from it via a temporary network partition. This also happens upon bootup, or whenever the *File Server* must throw away its record of a *Cache Manager* because its tables have been filled.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`---` No error codes are generated.

6.5.5.4 RXAFSCB_GetLock — Get contents of *Cache Manager* lock table

```
int RXAFSCB_GetLock(IN struct rx_call *a_rxCall,
                    IN long a_index,
                    OUT AFSDBLock *a_lockP)
```

Description

[Opcode 207] Fetch the contents of entry *a_index* in the *Cache Manager* lock table. There are 11 locks in the table, as described in Section 6.5.2. The contents of the desired lock, including a string name representing the lock, are returned in *a_lockP*.

This call is not used by *File Server*s, but rather by debugging tools such as `cmdebug`.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

1 The index value supplied in *a_index* is out of range; it must be between 0 and 10.

6.5.5.5 RXAFSCB_GetCE — Get cache file description

```
int RXAFSCB_GetCE(IN struct rx_call *a_rxCall,
                  IN long a_index,
                  OUT AFSDBCacheEntry *a_ceP)
```

Description

[Opcode 208] Fetch the description for entry *a_index* in the *Cache Manager* file cache, storing it into the buffer to which *a_ceP* points. The structure returned into this pointer variable is described in Section 4.3.2.

This call is not used by *File Server*s, but rather by debugging tools such as `cmdebug`.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`1` The index value supplied in *a_index* is out of range.

6.5.5.6 RXAFSCB_XStatsVersion — Get version of extended statistics package

```
int RXAFSCB_XStatsVersion(IN struct rx_call *a_rxCall,
                          OUT long *a_versionNumberP)
```

Description

[Opcode 209] This call asks the *Cache Manager* for the current version number of the extended statistics structures it exports (see *RXAFSCB_GetXStats()*, Section 6.5.5.7). The version number is placed in *a_versionNumberP*.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`---` No error codes are generated.

6.5.5.7 RXAFSCB_GetXStats — Get contents of extended statistics data collection

```
int RXAFSCB_GetXStats(IN struct rx_call *a_rxCall,
                      IN long a_clientVersionNumber,
                      IN long a_collectionNumber,
                      OUT long *a_srvVersionNumberP,
                      OUT long *a_timeP,
                      OUT AFSCB_CollData *a_dataP)
```

Description

[Opcode 210] This function fetches the contents of the specified *Cache Manager* extended statistics structure. The caller provides the version number of the data it expects to receive in *a_clientVersionNumber*. Also provided in *a_collectionNumber* is the numerical identifier for the desired **data collection**. There are currently two of these data collections defined: `AFSCB_XSTATSCOLL_CALL_INFO`, which is the list of tallies of the number of invocations of internal *Cache Manager* procedure calls, `AFSCB_XSTATSCOLL_PERF_INFO`, which is a list of performance-related numbers, and `AFSCB_XSTATSCOLL_FULL_PERF_INFO`, which is a list of extended performance-related numbers. The precise contents of these collections are described in Section 6.5.4. The current version number of the *Cache Manager* collections is returned in *a_srvVersionNumberP*, and is always set upon return, even if the caller has asked for a different version. If the correct version number has been specified, and a supported collection number given, then the collection data is returned in *a_dataP*. The time of collection is also returned, being placed in *a_timeP*.

Rx call information for the related *Cache Manager* is contained in *a_rxCallP*.

Error Codes

`1` The collection number supplied in *a_collectionNumber* is out of range.

6.6 Files

The *Cache Manager* gets some of its start-up configuration information from files located on the client machine's hard disk. Each client is required to supply a */usr/vice/etc* directory in which this configuration data is kept. Section 6.6.1 describes the format and purpose of the three files contributing this setup information: *ThisCell*, *CellServDB*, and *cacheinfo*.

6.6.1 Configuration Files

6.6.1.1 *ThisCell*

The *Cache Manager*, along with various applications, needs to be able to determine the cell to which its client machine belongs. This information is provided by the *ThisCell* file. It contains a single line stating the machine's fully-qualified cell name.

As with the *CellServDB* configuration file, the *Cache Manager* reads the contents of *ThisCell* exactly once, at start-up time. Thus, an incarnation of the *Cache Manager* will maintain precisely one notion of its home cell for its entire lifetime. Thus, changes to the text of the *ThisCell* file will be invisible to the running *Cache Manager*. However, these changes will affect such application programs as `klog`, which allows a user to generate new authentication tickets. In this example, `klog` reads *ThisCell* every time it is invoked, and then interacts with the set of *Authentication Server*s running in the given home cell, unless the caller specifies the desired cell on the command line.

The *ThisCell* file is not expected to be changed on a regular basis. Client machines are not imagined to be frequently traded between different administrative organizations. The Unix mode bits are set to specify that while everyone is allowed to read the file, only `root` is allowed to modify it.

6.6.1.2 *CellServDB*

To conduct business with a given AFS cell, a *Cache Manager* must be informed of the cell's name and the set of machines running AFS database servers within that cell. Such servers include the *Volume Location Server*, *Authentication Server*, and *Protection Server*. This particular cell information is obtained upon startup by reading the *CellServDB* file. Thus, when the *Cache Manager* initialization is complete, it will be able to communicate with the cells covered by *CellServDB*.

The following is an excerpt from a valid *CellServDB* file, demonstrating the format used.

```
  .
  .
  .
>transarc.com          #Transarc Corporation
192.55.207.7                   #henson.transarc.com
192.55.207.13                  #bigbird.transarc.com
192.55.207.22                  #ernie.transarc.com
>andrew.cmu.edu        #Carnegie Mellon University
128.2.10.2                     #vice2.fs.andrew.cmu.edu
128.2.10.7                     #vice7.fs.andrew.cmu.edu
128.2.10.10                    #vice10.fs.andrew.cmu.edu
  .
  .
  .
```

There are four rules describing the legal *CellServDB* file format:

1. Each cell has a separate entry. The entries may appear in any order. It may be convenient, however, to have the workstation's local cell be the first to appear.
2. No blank lines should appear in the file, even at the end of the last entry.
3. The first line of each cell's entry begins with the ">" character, and specifies the cell's human-readable, Internet Domain-style name. Optionally, some white space and a comment (preceded by a "#") may follow, briefly describing the specified cell.
4. Each subsequent line in a cell's entry names one of the cell's database server machines. The following must appear on the line, in the order given:
   - The Internet address of the server, in the standard 4-component dot notation.
   - Some amount of whitespace.
   - A "#", followed by the machine's complete Internet host name. **In this instance, the "#" sign and the text beyond it specifying the machine name are NOT treated as a comment. This is required information.** The *Cache Manager* will use the given host name to determine its current address via an Internet Domain lookup. If and only if this lookup fails does the *Cache Manager* fall back to using the dotted Internet address on the first part of the line. This dotted address thus appears simply as a hint in case of Domain database downtime.

The *CellServDB* file is only parsed once, when the *Cache Manager* first starts. It is possible, however, to amend existing cell information records or add completely new ones at any time after *Cache Manager* initialization completes. This is accomplished via the `VIOCNEWCELL` **pioctl()** (see Section 6.4.5.1.

6.6.1.3 *cacheinfo*

This one-line file contains three fields separated by colons:

- **AFS Root Directory**: This is the directory where the *Cache Manager* mounts the AFS root volume. Typically, this is specified to be */afs*.
- **Cache Directory**: This field names the directory where the *Cache Manager* is to create its local cache files. This is typically set to */usr/vice/cache*.
- **Cache Blocks**: The final field states the upper limit on the number of 1,024-byte blocks that the *Cache Manager* is allowed to use in the partition hosting the named cache directory.

Thus, the following `cacheinfo` file would instruct the *Cache Manager* to mount the AFS filespace at */afs*, and inform it that it may expect to be able to use up to 25,000 blocks for the files in its cache directory, */usr/vice/cache*.

```
/afs:/usr/vice/cache:25000
```

6.6.2 Cache Information Files

6.6.2.1 *AFSLog*

This is the AFS log file used to hold *Cache Manager* debugging output. The file is set up when the *Cache Manager* first starts. If it already exists, it is truncated. If it doesn't, it is created. Output to this file is enabled and disabled via the the `VIOC_VENUSLOG` **pioctl()** (see Section 6.4.9.2). Normal text messages are written to this file by the *Cache Manager* when output is enabled. Each time logging to this file is enabled, the *AFSLog* file is truncated. Only `root` can read and write this file.

6.6.2.2 *CacheItems*

The *Cache Manager* only keeps a subset of its data cache entry descriptors in memory at once. The number of these in-memory descriptors is determined by *afsd*. All of the data cache entry descriptors are kept on disk, in the *CacheItems* file. The file begins with a header region, taking up four longwords:

```
struct fheader {
long magic AFS_FHMAGIC   0x7635fab8
long firstCSize: First chunk size
long otherCSize: Next chunk sizes
long spare
}
```

The header is followed by one entry for each cache file. Each is:

```
struct fcache {
    short hvNextp;              /* Next in vnode hash table, or freeDCList */
    short hcNextp;              /* Next index in [fid, chunk] hash table */
    short chunkNextp;           /* File queue of all chunks for a single vnode */
    struct VenusFid fid;        /* Fid for this file */
    long modTime;               /* last time this entry was modified */
    long versionNo;             /* Associated data version number */
    long chunk;                 /* Relative chunk number */
    long inode;                 /* Unix inode for this chunk */
    long chunkBytes;            /* Num bytes in this chunk */
    char states;                /* Has this chunk been modified? */
};
```

6.6.2.3 *VolumeItems*

The *Cache Manager* only keeps at most `MAXVOLS` (50) in-memory volume descriptions. However, it records all volume information it has obtained in the *VolumeItems* file in the chosen AFS cache directory. This file is truncated when the *Cache Manager* starts. Each volume record placed into this file has the following `struct fvolume` layout:

```
struct fvolume {
    long cell;                  /*Cell for this entry*/
    long volume;                /*Numerical volume ID */
    long next;                  /*Hash index*/
    struct VenusFid dotdot;     /*Full fid for .. dir */
    struct VenusFid mtpoint;    /*Full fid for mount point*/
};
```

6.7 Mariner Interface

The *Cache Manager* Mariner interface allows interested parties to be advised in real time as to which files and/or directories are being actively transferred between the client machine and one or more *File Server*s. If enabled, this service delivers messages of two different types, as exemplified below:

```
Fetching myDataDirectory
Fetching myDataFile.c
Storing myDataObj.o
```

In the first message, the *myDataDirectory* directory is shown to have just been fetched from a *File Server*. Similarly, the second message indicates that the C program *myDataFile.c* had just been fetched from its *File Server* of residence. Finally, the third message reveals that the *myDataObj.o* object file has just been written out over the network to its respective server.

In actuality, the text of the messages carries a string prefix to indicate whether a Fetch or Store operation had been performed. So, the full contents of the above messages are as follows:

```
fetch$Fetching myDataDirectory
fetch$Fetching myDataFile.c
store$Storing myDataObj.o
```

The Mariner service may be enabled or disabled for a particular machine by using the `VIOC_AFS_MARINER_HOST` **pioctl()** (see Section 6.4.9.1). This operation allows any host to be specified as the recipient of these messages. A potential recipient must have its host be declared the target of such messages, then listen to a socket on port 2106.

Internally, the *Cache Manager* maintains a cache of `NMAR` (10) vnode structure pointers and the string name (up to 19 characters) of the associated file or directory. This cache is implemented as an array serving as a circular buffer. Each time a file is involved in a create or lookup operation on a *File Server*, the current slot in this circular buffer is filled with the relevant vnode and string name information, and the current position is advanced. If Mariner output is enabled, then an actual network fetch or store operation will trigger messages of the kind shown above. Since a fetch or store operation normally occurs shortly after the create or lookup, the mapping of vnode to name is likely to still be in the Mariner cache when it comes time to generate the appropriate message. However, since an unbounded number of other lookups or creates could have been performed in the interim, there is no guarantee that the mapping entry will not have been overrun. In these instances, the Mariner message will be a bit vaguer. Going back to our original example,

```
Fetching myDataDirectory
Fetching a file
Storing myDataObj.o
```

In this case, the cached association between the vnode containing *myDataFile.c* and its string name was thrown out of the Mariner cache before the network fetch operation could be performed. Unable to find the mapping, the generic phrase "a file" was used to identify the object involved.

Mariner messages only get generated when RPC traffic for fetching or storing a file system object occurs between the *Cache Manager* and a *File Server*. Thus, file accesses that are handled by the *Cache Manager*'s on-board data cache do not trigger such announcements.

Chapter 7

struct afs_CMCallStats

This appendix contains the full specification of the `afs_CMCallStats` structure, as originally described in Section 6.5.4.2. This construct is returned by the *Cache Manager* when a client requests data collection `AFSCB_XSTATSCOLL_CALL_INFO`, as defined in Section 6.5.3.

In the following tabular presentation, each field name listed is preceded by its (longword) offset in the `afs_CMCallStats` structure.

0. C_afs_init	1. C_gop_rdwr	2. C_aix_gnode_rele
3. C_gettimeofday	4. C_m_cpytoc	5. C_aix_vattr_null
6. C_afs_gn_ftrunc	7. C_afs_gn_rdwr	8. C_afs_gn_ioctl
9. C_afs_gn_lockctl	10. C_afs_gn_readlink	11. C_afs_gn_readdir
12. C_afs_gn_select	13. C_afs_gn_strategy	14. C_afs_gn_symlink
15. C_afs_gn_revoke	16. C_afs_gn_link	17. C_afs_gn_mkdir
18. C_afs_gn_mknod	19. C_afs_gn_remove	20. C_afs_gn_rename
21. C_afs_gn_rmdir	22. C_afs_gn_fid	23. C_afs_gn_lookup
24. C_afs_gn_open	25. C_afs_gn_create	26. C_afs_gn_hold
27. C_afs_gn_close	28. C_afs_gn_map	29. C_afs_gn_rele
30. C_afs_gn_unmap	31. C_afs_gn_access	32. C_afs_gn_getattr
33. C_afs_gn_setattr	34. C_afs_gn_fclear	35. C_afs_gn_fsync
36. C_pHash	37. C_DInit	38. C_DRead
39. C_FixupBucket	40. C_afs_newslot	41. C_DRelease
42. C_DFlush	43. C_DFlushEntry	44. C_DVOffset
45. C_DZap	46. C_DNew	47. C_shutdown_bufferpackage
48. C_afs_CheckKnownBad	49. C_afs_RemoveVCB	50. C_afs_NewVCache
51. C_afs_FlushActiveVcaches	52. C_afs_VerifyVCache	53. C_afs_WriteVCache
54. C_afs_GetVCache	55. C_afs_StuffVcache	56. C_afs_FindVCache
57. C_afs_PutDCache	58. C_afs_PutVCache	59. C_CacheStoreProc
60. C_afs_FindDCache	61. C_afs_TryToSmush	62. C_afs_AdjustSize
63. C_afs_CheckSize	64. C_afs_StoreWarn	65. C_CacheFetchProc

66. C_UFS_CacheStoreProc
67. C_UFS_CacheFetchProc
68. C_afs_GetDCache
69. C_afs_SimpleVStat
70. C_afs_ProcessFS
71. C_afs_InitCacheInfo
72. C_afs_InitVolumeInfo
73. C_afs_InitCacheFile
74. C_afs_CacheInit
75. C_afs_GetDSlot
76. C_afs_WriteThroughDSlots
77. C_afs_MemGetDSlot
78. C_afs_UFSGetDSlot
79. C_afs_StoreDCache
80. C_afs_StoreMini
81. C_shutdown_cache
82. C_afs_StoreAllSegments
83. C_afs_InvalidateAllSegments
84. C_afs_TruncateAllSegments
85. C_afs_CheckVolSync
86. C_afs_wakeup
87. C_afs_CFileOpen
88. C_afs_CFileTruncate
89. C_afs_GetDownD
90. C_afs_WriteDCache
91. C_afs_FlushDCache
92. C_afs_GetDownDSlot
93. C_afs_FlushVCache
94. C_afs_GetDownV
95. C_afs_QueueVCB
96. C_afs_call
97. C_afs_syscall_call
98. C_syscall
99. C_lpioctl
100. C_lsetpag
101. C_afs_syscall
102. C_afs_CheckInit
103. C_afs_shutdown
104. C_shutdown_BKG
105. C_shutdown_afstest
106. C_SRXAFSCB_GetCE
107. C_ClearCallBack
108. C_SRXAFSCB_GetLock
109. C_SRXAFSCB_CallBack
110. C_SRXAFSCB_InitCallBackState
111. C_SRXAFSCB_Probe
112. C_afs_RXCallBackServer
113. C_shutdown_CB
114. C_afs_Chunk
115. C_afs_ChunkBase
116. C_afs_ChunkOffset
117. C_afs_ChunkSize
118. C_afs_ChunkToBase
119. C_afs_ChunkToSize
120. C_afs_SetChunkSize
121. C_afs_config
122. C_mem_freebytes
123. C_mem_getbytes
124. C_fpalloc
125. C_kluge_init
126. C_ufdalloc
127. C_ufdfree
128. C_commit
129. C_dev_ialloc
130. C_ffree
131. C_iget
132. C_iptovp
133. C_ilock
134. C_irele
135. C_iput
136. C_afs_Daemon
137. C_afs_CheckRootVolume
138. C_BPath
139. C_BPrefetch
140. C_BStore
141. C_afs_BBusy
142. C_afs_BQueue
143. C_afs_BRelease
144. C_afs_BackgroundDaemon
145. C_shutdown_daemons
146. C_exporter_add
147. C_exporter_find
148. C_afs_gfs_kalloc
149. C_IsAfsVnode
150. C_SetAfsVnode
151. C_afs_gfs_kfree
152. C_gop_lookupname
153. C_gfsvop_getattr
154. C_gfsvop_rdwr
155. C_afs_uniqtime
156. C_gfs_vattr_null
157. C_afs_lock
158. C_afs_unlock
159. C_afs_update
160. C_afs_gclose
161. C_afs_gopen
162. C_afs_greadlink
163. C_afs_select
164. C_afs_gbmap
165. C_afs_getfsdata
166. C_afs_gsymlink
167. C_afs_namei
168. C_printgnode
169. C_HaveGFSLock
170. C_afs_gmount
171. C_AddGFSLock
172. C_RemoveGFSLock
173. C_afs_grlock
174. C_afs_gumount
175. C_afs_gget
176. C_afs_glink
177. C_afs_gmkdir
178. C_afs_sbupdate
179. C_afs_unlink
180. C_afs_grmdir
181. C_afs_makenode
182. C_afs_grename
183. C_afs_rele
184. C_afs_syncgp
185. C_afs_getval
186. C_afs_gfshack
187. C_afs_trunc
188. C_afs_rwgp
189. C_afs_stat
190. C_afsc_link
191. C_hpsobind
192. C_hpsoclose
193. C_hpsocreate
194. C_hpsoreserve
195. C_afs_vfs_mount
196. C_devtovfs
197. C_igetinode
198. C_afs_syscall_iopen
199. C_iopen
200. C_afs_syscall_iincdec
201. C_afs_syscall_ireadwrite
202. C_iincdec
203. C_ireadwrite
204. C_oiread
205. C_AHash
206. C_QTOA
207. C_afs_FindPartByDev
208. C_aux_init
209. C_afs_GetNewPart
210. C_afs_InitAuxVolFile
211. C_afs_CreateAuxEntry
212. C_afs_GetAuxSlot
213. C_afs_GetDownAux
214. C_afs_FlushAuxCache
215. C_afs_GetAuxInode

struct afs_CMCallStats

216. C_afs_PutAuxInode
217. C_afs_ReadAuxInode
218. C_afs_WriteAuxInode
219. C_afs_auxcall
220. C_tmpdbg_auxtbl
221. C_tmpdbg_parttbl
222. C_idec
223. C_iinc
224. C_iread
225. C_iwrite
226. C_getinode
227. C_trygetfs
228. C_iforget
229. C_afs_syscall_icreate
230. C_icreate
231. C_Lock_Init
232. C_Lock_Obtain
233. C_Lock_ReleaseR
234. C_Lock_ReleaseW
235. C_afs_BozonLock
236. C_afs_BozonUnlock
237. C_osi_SleepR
238. C_osi_SleepS
239. C_osi_SleepW
240. C_osi_Sleep
241. C_afs_BozonInit
242. C_afs_CheckBozonLock
243. C_xxxinit
244. C_KernelEntry
245. C_afs_InitMemCache
246. C_afs_LookupMCE
247. C_afs_MemReadBlk
248. C_afs_MemReadUIO
249. C_afs_MemWriteBlk
250. C_afs_MemCacheStoreProc
251. C_afs_MemCacheTruncate
252. C_afs_MemWriteUIO
253. C_afs_MemCacheFetchProc
254. C_afs_vnode_pager_create
255. C_next_KernelEntry
256. C_afs_GetNfsClientPag
257. C_afs_FindNfsClientPag
258. C_afs_PutNfsClientPag
259. C_afs_nfsclient_reqhandler
260. C_afs_nfsclient_GC
261. C_afs_nfsclient_hold
262. C_afs_nfsclient_stats
263. C_afs_nfsclient_sysname
264. C_afs_nfsclient_shutdown
265. C_afs_rfs_readdir_fixup
266. C_afs_rfs_dispatch
267. C_afs_xnfs_svc
268. C_afs_xdr_putrddirres
269. C_afs_rfs_readdir
270. C_afs_rfs_rddirfree
271. C_rfs_dupcreate
272. C_rfs_dupsetattr
273. C_Nfs2AfsCall
274. C_afs_sun_xuntext
275. C_osi_Active
276. C_osi_FlushPages
277. C_osi_FlushText
278. C_osi_CallProc
279. C_osi_CancelProc
280. C_osi_Invisible
281. C_osi_Time
282. C_osi_Alloc
283. C_osi_SetTime
284. C_osi_Dump
285. C_osi_Free
286. C_shutdown_osi
287. C_osi_UFSOpen
288. C_osi_Close
289. C_osi_Stat
290. C_osi_Truncate
291. C_osi_Read
292. C_osi_Write
293. C_osi_MapStrategy
294. C_shutdown_osifile
295. C_osi_FreeSendSpace
296. C_osi_FreeSmallSpace
297. C_pkt_iodone
298. C_shutdown_osinet
299. C_afs_cs
300. C_osi_AllocSendSpace
301. C_osi_AllocSmallSpace
302. C_osi_CloseToTheEdge
303. C_osi_xgreedy
304. C_osi_FreeSocket
305. C_osi_NewSocket
306. C_trysblock
307. C_osi_NetSend
308. C_WaitHack
309. C_osi_CancelWait
310. C_osi_InitWaitHandle
311. C_osi_Wakeup
312. C_osi_Wait
313. C_dirp_Read
314. C_dirp_SetCacheDev
315. C_Die
316. C_dirp_Cpy
317. C_dirp_Eq
318. C_dirp_Write
319. C_dirp_Zap
320. C_PSetVolumeStatus
321. C_PFlush
322. C_PNewStatMount
323. C_PGetTokens
324. C_PUnlog
325. C_PCheckServers
326. C_PMariner
327. C_PCheckAuth
328. C_PCheckVolNames
329. C_PFindVolume
330. C_Prefetch
331. C_PGetCacheSize
332. C_PRemoveCallBack
333. C_PSetCacheSize
334. C_PViceAccess
335. C_PListCells
336. C_PNewCell
337. C_PRemoveMount
338. C_HandleIoctl
339. C__AFSIOCTL
340. C__VALIDAFSIOCTL
341. C_PGetCellStatus
342. C_PSetCellStatus
343. C_PVenusLogging
344. C_PFlushVolumeData
345. C_PSetSysName
346. C_PExportAfs
347. C_HandleClientContext
348. C_afs_ioctl
349. C_afs_xioctl
350. C_afs_pioctl
351. C_afs_syscall_pioctl
352. C_HandlePioctl
353. C_PGetAcl
354. C_PGetFID
355. C_PSetAcl
356. C_PBogus
357. C_PGetFileCell
358. C_PGetWSCell
359. C_PNoop
360. C_PGetUserCell
361. C_PSetTokens
362. C_PGetVolumeStatus
363. C_afs_ResetAccessCache
364. C_afs_FindUser
365. C_afs_ResetUserConns

struct afs_CMCallStats

366. C_afs_ResourceInit
367. C_afs_GetCell
368. C_afs_GetCellByIndex
369. C_afs_GetCellByName
370. C_afs_NewCell
371. C_afs_GetUser
372. C_afs_PutUser
373. C_afs_SetPrimary
374. C_CheckVLDB
375. C_afs_GetVolume
376. C_afs_GetVolumeByName
377. C_InstallVolumeEntry
378. C_InstallVolumeInfo
379. C_afs_FindServer
380. C_afs_PutVolume
381. C_afs_random
382. C_ranstage
383. C_RemoveUserConns
384. C_afs_MarinerLog
385. C_afs_vtoi
386. C_afs_GetServer
387. C_afs_SortServers
388. C_afs_Conn
389. C_afs_ConnByHost
390. C_afs_ConnByMHosts
391. C_afs_Analyze
392. C_afs_PutConn
393. C_afs_ResetVolumeInfo
394. C_StartLogFile
395. C_afs_SetLogFile
396. C_EndLogFile
397. C_afs_dp
398. C_fprf
399. C_fprint
400. C_fprintn
401. C_afs_CheckLocks
402. C_puttofile
403. C_shutdown_AFS
404. C_afs_CheckCacheResets
405. C_afs_GCUserData
406. C_VSleep
407. C_afs_CheckCode
408. C_afs_CopyError
409. C_afs_FinalizeReq
410. C_afs_cv2string
411. C_afs_FindVolCache
412. C_afs_GetVolCache
413. C_afs_GetVolSlot
414. C_afs_WriteVolCache
415. C_afs_UFSGetVolSlot
416. C_afs_CheckVolumeNames
417. C_afs_MemGetVolSlot
418. C_print_internet_address
419. C_CheckVLServer
420. C_HaveCallBacksFrom
421. C_ServerDown
422. C_afs_CheckServers
423. C_afs_AddToMean
424. C_afs_GetCMStat
425. C_afs_getpage
426. C_afs_putpage
427. C_afs_nfsrdwr
428. C_afs_map
429. C_afs_cmp
430. C_afs_cntl
431. C_afs_dump
432. C_afs_realvp
433. C_afs_PageLeft
434. C_afsinit
435. C_afs_mount
436. C_afs_unmount
437. C_afs_root
438. C_afs_statfs
439. C_afs_sync
440. C_afs_vget
441. C_afs_mountroot
442. C_afs_swapvp
443. C_afs_AddMarinerName
444. C_afs_setpag
445. C_genpag
446. C_getpag
447. C_afs_GetMariner
448. C_afs_badop
449. C_afs_index
450. C_afs_noop
451. C_afs_open
452. C_afs_closex
453. C_afs_close
454. C_afs_MemWrite
455. C_afs_write
456. C_afs_UFSWrite
457. C_afs_rdwr
458. C_afs_MemRead
459. C_afs_read
460. C_FIXUPSTUPIDINODE
461. C_afs_UFSRead
462. C_afs_CopyOutAttrs
463. C_afs_getattr
464. C_afs_VAttrToAS
465. C_afs_setattr
466. C_EvalMountPoint
467. C_afs_access
468. C_ENameOK
469. C_HandleAtName
470. C_getsysname
471. C_strcat
472. C_afs_lookup
473. C_afs_create
474. C_afs_LocalHero
475. C_FetchWholeEnchilada
476. C_afs_remove
477. C_afs_link
478. C_afs_rename
479. C_afs_InitReq
480. C_afs_mkdir
481. C_BlobScan
482. C_afs_rmdir
483. C_RecLen
484. C_RoundToInt
485. C_afs_readdir_with_offlist
486. C_DIRSIZ_LEN
487. C_afs_readdir_move
488. C_afs_readdir_iter
489. C_HandleFlock
490. C_afs_readdir
491. C_afs_symlink
492. C_afs_HandleLink
493. C_afs_MemHandleLink
494. C_afs_UFSHandleLink
495. C_afs_readlink
496. C_afs_fsync
497. C_afs_inactive
498. C_afs_ustrategy
499. C_afs_bread
500. C_afs_brelse
501. C_afs_bmap
502. C_afs_fid
503. C_afs_strategy
504. C_afs_FakeClose
505. C_afs_FakeOpen
506. C_afs_StoreOnLastReference
507. C_afs_GetAccessBits
508. C_afs_AccessOK
509. C_shutdown_vnodeops
510. C_afsio_copy
511. C_afsio_trim
512. C_afs_page_read
513. C_afs_page_write
524. C_afsio_skip
515. C_afs_read1dir

struct afs_CMCallStats

516. C_afs_get_groups_from_pag
517. C_afs_get_pag_from_groups
518. C_PagInCred
519. C_afs_getgroups
520. C_setpag
521. C_afs_setgroups
522. C_afs_page_in
523. C_afs_page_out
524. C_AddPag
525. C_afs_AdvanceFD
526. C_afs_lockf
527. C_afs_xsetgroups
528. C_afs_nlinks
529. C_DoLockWarning
530. C_afs_lockctl
531. C_afs_xflock

struct afs_CMCallStats

Bibliography

[1] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[2] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[3] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[4] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*, RFC 1094, March 1989.

[5] Sun Microsystems, Inc. *Design and Implementation of the Sun Network File System*, USENIX Summer Conference Proceedings, June 1985.

[6] S.P. Miller, B.C. Neuman, J.I. Schiller, J.H. Saltzer. *Kerberos Authentication and Authorization System*, Project Athena Technical Plan, Section E.2.1, M.I.T., December 1987.

[7] Bill Bryant. *Designing an Authentication System: a Dialogue in Four Scenes*, Project Athena internal document, M.I.T, draft of 8 February 1988.

struct afs_CMCallStats

Protection Server Interface

Chapter 1

Overview

1.1 Introduction

In AFS, protection of files is accomplished by the use of Access Control Lists (ACLs) at the directory level. It is undesirable to allow ACLs to grow without reasonable bounds, as the ACLs themselves are stored as part of the directory they govern. Encoding the full string names of the individual users listed in the ACL is clearly a waste of space, so their numerical identities are stored instead. Also, it was deemed useful to allow the use of a single identity to specify a *group* of users, all of whom share the identical access to the files in the associated directory.

The *Protection Server* manages the AFS **Protection Database**, or **PDB**. This database maintains information on individual AFS users as well as groups of related AFS users. The *Protection Server* provides the service whereby properly-authorized AFS agents may reference and manipulate this database. Specifically, it exports an RPC interface through which callers may:

- Map back and forth between printable user/group names and their equivalent numerical identifiers.
- Query the information held for any given AFS user or group.
- Create, modify, and delete the records in the PDB where the above information is held.
- Generate a list of all identities with which a given individual user is associated. Along with the user's login name, this list consists of all AFS groups to which they

belong. This list is referred to as the user's **Current Protection Subdomain**, or **CPS**.

1.2 Scope

This paper describes the design and structure of the AFS *Protection Server*. The scope of this work is to provide readers with a sufficiently detailed description of the *Protection Server* so that they may construct client applications that call the server's RPC interface routines.

1.3 Document Layout

The second chapter discusses various aspects of the *Protection Server*'s architecture. First, the notions and characteristics of AFS **users** and **groups** are covered. The set of built-in AFS groups is then described. Next, the important notion of a **Current Protection Subdomain (CPS)**, is examined. Two unimplemented yet externally visible *Protection Server* features are discussed, namely "foreign" users and superset/subset instances of individual users. Following this, the makeup of the **Protection Database (PDB)** managed by the *Protection Server* is presented. Finally, the *ptserver* process that implements the *Protection Server* is explored.

The third and final chapter provides a detailed examination of the programmer-visible *Protection Server* constants and structures, along with a full specification of the API for the RPC-based *Protection Server* functionality.

1.4 Related Documents

This document is a member of a documentation suite providing programmer-level specifications as to the operation and interfaces offered by the various AFS servers and agents, as well as the underlying RPC system they use to communicate. The full suite of related AFS specification documents is listed below:

- *AFS-3 Programmer's Reference: Architectural Overview*: This paper provides an architectual overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and examining their interactions.

- *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*: This document describes the *File Server* and *Cache Manager* agents, which provide the backbone file managment services for AFS. The collection of *File Server*s for a cell supply centralized file storage for that site, and allow clients running the *Cache Manager* component to acces those files in a high-performance, secure fashion.
- *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.
- *AFS-3 Programmer's Reference: BOS Server Interface*: This paper explains the "nanny" service for AFS server agents, supervising and controlling their execution as well as handling deployment of new server software. The *BOS Server* assists in the administrability of the AFS environment.
- *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*: This document specifies the design and operation of the remote procedure call and lightweight process packages used by AFS.

In addition to these papers, the AFS 3.1 product is delivered with its own user, administrator, installation, and command reference documents.

Chapter 2

Protection Server **Architecture**

This chapter is responsible for presenting the general architecture of the *Protection Server*. First, the building blocks it works with, the notions of **user** and **group**, are examined in detail. Following this, the nature and organization of the **Protection Database (PDB)** is discussed. Finally, the makeup and command line arguments of the *ptserver* process itself are covered.

2.1 Building Blocks: Users and Groups

2.1.1 Introduction and References

A detailed treatment of the AFS notions of *user* and *group* is available in the chapter entitled *The Protection Database* in the *AFS System Administrator's Guide* [1]. This section will serve to present its main points, focus on certain aspects, and amplify on certain others. Readers wishing to get a fuller picture are encouraged to study the given citation.

Users and groups are the two basic entities tracked and manipulated by the *Protection Server* as it maintains the PDB. A user is simply an individual who is capable of authenticating to the local AFS cell. A group is a list of individual users. In particular, a group may *not* have another group as one of its members.

Users and groups may both be dynamically created and deleted within AFS. While only administratively-qualified people may create new users, any authenticated individual may create groups without possessing any special system privileges. Each AFS user is granted a **group creation quota**, placing an upper limit on the number of groups they may generate at any one time.

All AFS user identifiers in the PDB are denoted by positive integers, and all group identifiers by negative integers. Zero is a reserved value in this domain.

2.1.2 User and Group Names

Names for individual AFS users are those strings typed to the `login` and `klog` programs, and are the string representations of the user's account with AFS. The *Protection Server* transforms all names to lower-case for storage and comparison purposes. Names for AFS groups fall into the following two categories:

- **Prefixed names**: These group names are composed of two distinct sections, separated by a colon. The prefix, or part before the colon, identifies the user or group marked as the *owner* of the group. The latter part of the name identifies the name of the group proper, as chosen by its creator.
- **Non-prefixed names**: Groups whose names lack the prefix portion described above (and the associated colon) may also be created. These are typically used when creating other normal prefixed group names for which a "generic" owner is desired. For example, a non-prefixed group named *accounting* may be created, and populated with the administrators of a company's accounting department. This allows for the creation of a group called *accounting:payable* created to handle the files within those directories controlled by the Accounts Payable personnel. Since this new group is owned by all members of the *accounting* group, any one of them may administer this group

2.1.3 Built-In Groups

When the *Protection Server* initializes a new PDB, it automatically creates entries for the following three specialized groups:

- **system:anyuser**: Any individual who accesses AFS files is considered by the system to be a member of this group, whether or not they are authenticated within the given cell. This group is unusual in that it doesn't have a stable membership. In fact, it doesn't have an explicit list of members. Instead, the `system:anyuser` "membership" grows and shrink as file accesses occur, with users being (conceptually) added and deleted automatically as they interact with the system. Because of the way the `system:anyuser` group is implemented by the *Protection Server*, with its non-standard representation and dynamic membership changes, an application may *not* use the standard *Protection Server* interface operations to manipulate this group.

The `system:anyuser` group is typically put on the ACL of those directories for which some specific level of completely public access is desired, covering anyone at any site in the greater AFS community.

- **system:authuser**: Any individual in possession of a valid Kerberos ticket minted by the cell's *Authentication Server* is treated as a member of this group. Just as with `system:anyuser`, this special group does not have a stable membership, and its members are not explicitly added and deleted via calls to the *Protection Server* interface. If you acquire a ticket within the cell, you are automatically "added" to the group. If the ticket expires or it gets thrown away by the user via an `unlog` operation, then the given individual will automatically be "removed" from the group.

  The `system:authuser` group is usually put on the ACL of those directories for which some specific level of intra-site access is desired. Anyone holding a valid ticket within the cell will be allowed to perform the set of accesses specified by the ACL entry, regardless of their precise individual ID.

- **system:administrators**: This built-in group defines the set of users capable of performing certain important administrative operations within the cell. Members of this group have explicit "a" (Administer) rights on every directory's ACL in the cell. Members of the `system:administrators` group, for example, are the only ones capable of issuing privileged commands within the `pts` utility, which invokes the functions defined in the *Protection Server* RPC interface (described in detail in Section 3.6). Similarly, only `system:administrators` members can use privileged `fs` commands, making them capable of changing volume quotas. This group is *not* like the other two described above in that it *does* have a stable membership. Individuals must be added to and deleted from this group explicitly with the appropriate `pts` commands.

  The `system:administrators` group is typically put on the ACL of those directories which contain sensitive administrative information, or on those places where only administrators are allowed to make changes. All members of this group have implicit rights to change the ACL on any AFS directory within their cell. Thus, they don't have to actually appear on an ACL, or have "a" rights enabled in their ACL entry if they do appear, to be able to modify the ACL.

2.1.4 Current Protection Subdomain (CPS)

When a user attempts to perform an operation on an AFS file, the *File Server* receiving the request must determine whether the caller has adequate access privileges. Specifically, it must collect the complete set of identities associated with the caller and then compare them against the set listed in the appropriate ACL. If the *File Server* discovers that the caller or one of the groups of which he is a member appears in the ACL with the necessary rights, then the operation is honored. Should such an entry not be found, then the operation is rejected with a protection violation error.

The complete set of identities with which a user is associated, namely the user's own numerical ID, the set of groups to which the user belongs, along with the special identities available to AFS principals (e.g., `system:anyuser` is referred to as the **Current Protection Subdomain** or **CPS**. An individual user's CPS may change dynamically in response to the creation and deletion of groups in which they are members.

The *File Server* calculates a caller's CPS on a per-connection basis by calling the *Protection Server*. When the first call on a given *Rx* connection arrives, the *File Server* calculates the requestor's CPS and stores it with the caller's records. Note, however, that there is *no* callback mechanism that applies to the *File Server*'s view of CPS information. Should a user be added to or removed from a group, the *File Server*s that have already calculated that user's CPS in response to previous calls from that user are not informed of the update, and thus are left with stale information in these cases.

There are two ways that these caches may be brought up to date. If callers become quiescent for some period of time, their records will be garbage-collected by the *File Server*. The next file system operation from one of these users will generate a new *File Server* record containing the recalculated CPS. This correction may take longer than the caller is willing to wait, however (on the order of two hours). The second method is more direct, resulting in an immediate CPS recomputation. The user need simply re-authenticate with the `klog` utility program. This will generate a new (i.e., updated) Kerberos ticket representing the user's identity. The first new *Rx* call made by this user will result in a new *Rx* connection, which in turn creates a new *File Server* client entry for the user complete with a freshly-determined CPS.

2.1.5 Foreign Users

In the original *Protection Server* design, it was envisioned that support for providing name-to-ID mappings for authenticated "foreign" users (i.e., those users holding Kerberos tickets generated by *Authentication Server*s in a different cell) would be provided. As envisioned, numerical AFS IDs would be allocated from the local PDB as needed by users as these principals were added to ACLs. Thus, access control would be extended to include handling these foreign identities.

This feature was never implemented, yet hooks were left throughout the *Protection Server* data structures and code for its future support. Some of these hooks are listed below.

- `PRFOREIGN`: The `PRFOREIGN` constant was meant to identify a PDB entry as belonging to a foreign user.
- `PRCELL`: The `PRCELL` constant was meant to identify a PDB entry defining the characteristics of a particular foreign cell.
- `PR_SF_NUSERS`: The `PR_SF_NUSERS` constant is used in the `mask` argument to the *PR_SetFieldsEntry()* interface routine when the caller wishes to modify the PDB entry recording the quota an individual is granted for creating foreign user identities.
- `maxforeign`: This is a field in the PDB header (see the `struct prheader` description in Section 3.3.1) which records the largest foreign ID allocated to date.
- `foreigncount`: This is another field in the PDB header, and its job is to record the current number of foreign user entries.
- `nusers`: This is a field in the PDB entry data structure (see the `struct prentry` description in Section 3.3.2) which keeps track of the number of foreign user entries a particular individual has created. Note that there is also an `nusers` field in the user-visible version of the PDB entry data structure, `struct prdebugentry`.

2.1.6 Superset and Subset Instances

Another concept in the original *Protection Server* design that remains unimplemented in the current code is that of subset and superset instances of individual user identities. It seemed useful to allow for the dynamic creation of identities with restricted or amplified sets of access rights. For example, consider user `herbert`, a member of the `system:administrators` group. It would be possible for `herbert` to authenticate as a non-priviledged sub-identity, say `herbert.fewprivs`, as long as those administrative powers were not needed. This would insure that potentially destructive administrative operations would not be accidentally executed.

As with the unimplemented foreign user concept discussed in Section 2.1.5, there are a number of hooks in the *Protection Server* structures and code in support of the superset/subset notion. Some of these hooks are listed below.

- `PRINST`: This constant is used to mark a PDB entry as describing a superset or subset instance for a given AFS user.
- `maxInst`: The `maxInst` field in the PDB header (see the `struct prheader` description in Section 3.3.1) records the largest superset or subset ID created to date.
- `instcount`: This is another field in the PDB header, and it is responsible for recording the number of superset and subset instances currently registered in the system.
- `instance`: This is a field in the PDB entry data structure (see the `struct prentry` description in Section 3.3.2) which keeps track of the number of superset and subset instance entries a particular individual has created. Note that there is also an `instance` field in the user-visible version of the PDB entry data structure, `struct prdebugentry`.

2.2 The Protection Database

2.2.1 Layout

The organization of the Protection Database (PDB) is straightforward. At the beginning of the PDB is a header region, and it is immediately followed by the set of entry records for the users and groups in the system.

PDB entries are organized via two hash tables and several linked lists. Since PDB entries are normally looked up by either their string name or by their numerical ID, these entries are threaded into by-name and by-ID hash tables. There is a global free list maintained for deleted entries. There is also a global *orphan list*, which contains entries for all *orphans*, namely groups belonging to users that have been deleted from the PDB. In addition, each entry has a linked list containing the set of other entries which it owns.

The PDB entries store such information as the name and ID of the user or group, the IDs of its owner and its creator, the group creation quota, and a set of bit fields. These bits identify the entry's type (e.g., individual user, group), state whether quota enforcement and access checking is enabled for the entry, and specify the "privacy bits" for the entry (See Section 2.2.2). Also included in a PDB entry is a set of IDs with which the given entry is associated. If the entry is marked as representing an individual user, then the IDs identify the groups to which that individual belongs. If the entry is marked as representing a group, then the IDs identify the individual members belonging to that group.

The PDB entry described above can hold only `PRSIZE` (10) of these associated IDs. This number was considered reasonable for most occasions. However, it is possible for a user to belong to more than 10 AFS groups, or for a group to have more than 10 individuals as members. This is accomplished by introducing the concept of *continuation entries*. These continuation entries are exactly the same size as the above "regular" or "main" entries, so they are allocated, deleted, and threaded much like the main ones. However, one of the bit fields mentioned above is used to mark them as continuations. While they mirror some of the information in the main entry, continuation entries are formatted so that most of the space is devoted to a larger ID array, holding up to `COSIZE` (39) entries. Continuation entries are threaded to the appropriate main entry, and are thus used to hold the "spillover" IDs that don't fit there. An arbitrary number of continuation entries may be created for each AFS user or group represented in the PDB.

The PDB header, a `struct prheader` (as described in Section 3.3.1) contains such information as the PDB's version number, the address of the beginning of the free list, the number of total users and groups recorded for the system, the largest user and group IDs allocated so far, and the hash tables used for looking up entries by name and by ID.

2.2.2 PDB Entry Privacy Bits

It is possible to limit the kind of information that users obtain about entries in the PDB. This per-entry limitation is implemented by defining a set of *privacy bits* in each PDB record. Normal users are authorized to control external access to their own entries, as well as the set of group entries they own. Members of the privileged `system:administrators` group may set the privacy bits on any entry in the PDB.

There are five privacy bits, each denoted by a corresponding letter in the interface offered by the `pts` utility, which interacts with the *Protection Server*:

- `STATUS` **(s)**: Allows basic status for the entry itself to be seen.
- `OWNED` **(o)**: Allows the users and groups owned by the entry to be listed.
- `MEMBERSHIP` **(m)**: Allows the listing of the set of groups the individual user is a member of, or the set of users comprising the members of the given group.
- `ADD` **(a)**: Allows users to be added to the group.
- `REMOVE` **(r)**: Allows users to be removed from the group.

The `pts` utility sets up this list of rights to appear as five "slots", one for each of the above letters. Three variations of the appropriate letter may appear in each slot, each representing a different set of people who possess the right:

- **A hyphen** in the slot indicates that only the entry's owner and members of the `system:administrators` group have the access right. For user entries, this also indicates that the user has the right.
- **A lowercase letter** in the slot indicates that members of the group have the access right. This applies meaningfully to group entries only.
- **An uppercase letter** in the slot indicates that everyone has the access right. This is equivalent to the set of principals included in the `system:anyuser` group for ACLs.

Consider the example rights list `SOmar` associated with an AFS group entry. The two leading capital letters indicate that anyone can examine the group's entry and list the groups that it owns. The three following lowercase letters specify that only members of the group itself may list, add, or remove its members, respectively.

The default rights list associated with a user entry in the PDB is `S-----`. This specifies that anyone may examine the entry itself, but that only the user and members of the `system:administrators` group can list the groups they own and the groups of which they are members. The add and delete operations do not apply to a user entry.

The default rights list for a group entry is `S-M--`, meaning that all users can obtain status information on the group entry and list the members of the group.

2.2.3 Database Replication

The PDB, along with other important AFS databases, may be replicated to multiple sites to improve its availability. The *ubik* replication package is used to implement this functionality for the PDB. A full description of *ubik* and of the *quorum completion* algorithm it implements may be found in [4] and [5]. The basic abstraction provided by *ubik* is that of a disk file replicated to multiple server locations. One of the machines is considered to be the *synchronization site* at any given time, handling all write operations on the database file. Read operations may be directed to any of the active members of the *quorum*, namely a subset of the replication sites large enough to insure integrity across such failures as individual server crashes and network partitions. All of the quorum members participate in regular *elections* to determine the current synchronization site. Should the machine acting as the current synchronization site crash, another synchronization site will be elected. The *ubik* algorithms allow server machines to enter and exit the quorum in an orderly and consistent fashion. All operations to one of these replicated "abstract files" are performed as part of a *transaction*. If all the related operations performed under a transaction are successful, then the transaction is *committed*, and the changes are made permanent. Otherwise, the transaction is *aborted*, and all of the operations for that transaction are undone.

2.3 The *ptserver* Process

2.3.1 Introduction

The user-space *ptserver* process implements the *Protection Server*. This program maintains the PDB replica at the site where it runs, and cooperates with all other *ptserver* processes running in the cell to propagate updates to the replicated database. It implements the RPC interface defined in the *ptint.xg Rxgen* definition file. As part of its startup sequence, it must discover the PDB version it has on its local disk, move to join the quorum of replication sites for the PDB, and get the latest version if its local copy is out of date. Once the *ptserver* synchronizes with the other PDB replication sites, it will begin accepting *Protection Server* calls.

2.3.2 Command Line Switches

The *Protection Server* accepts a set of command line switches. Using these switches, a set of server parameters and actions can be set. Under normal conditions, the *Protection Server* process is started up by the *BOS Server* on that machine, as described in [1]. So, in order to utilize any of these command-line options, the system administrator must define the *Protection Server* record (bnode) in such a way that these parameters are properly included. Note that the switch names must be typed exactly as listed, and that abbreviations are not allowed.

Here is a description of the set of currently-supported command line switches.

- **-database** <*pathname*>: Provide the pathname prefix for the local PDB file to bring up. A synonym for the **-database** switch is **-db**. If this switch is not provided, the default database file prefix is set to */usr/afs/db/prdb*.

- **-rebuild**: Specify that the PDB is to be rebuilt before the given *Protection Server* begins offering its services. This command-line switch is currently a no-op.

2.3.3 Threading

As with all the other AFS server agents, the *Protection Server* is a multithreaded program. It is configured so that a minimum of two lightweight threads are guaranteed to be allocated to handle incoming RPC calls to the *Protection Server*, and a maximum of three threads are commissioned for this task.

Chapter 3

Protection Server **Interface**

3.1 Introduction

This chapter documents the API for the *Protection Server* facility, as defined by the *ptint.xg Rxgen* interface file and the *ptserver.h* include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer appear in this exposition.

3.2 Constants

This section covers the basic constant definitions of interest to the *Protection Server* application programmer. These definitions appear in the *ptint.h* file, automatically generated from the *ptint.xg Rxgen* interface file, and in *ptserver.h*.

Each subsection is devoted to describing the constants falling into each of the following categories:

- Interface function opcodes
- Size and configuration quantities
- Mask bits for *PR_SetFieldsEntry()*
- Reserved IDs
- Bits for the `struct prentry flags` field

- Privacy bits
- Miscellaneous

3.2.1 Interface Function Opcodes

These constants, appearing in the *ptopcodes.h Rxgen* interface file for the *Protection Server*, define the opcodes for the RPC routines. Every *Rx* call on this interface contains this opcode, and the thread dispatcher uses it to select the proper code at the server site to carry out the call.

Name	*Value*	*Description*
PRINEWUSER	500	Opcode for the *PR_NewUser()* function.
PRWHEREISIT	501	Opcode for the *PR_WhereIsIt()* function.
PRDUMPENTRY	502	Opcode for the *PR_DumpEntry()* function.
PRADDTOGROUP	503	Opcode for the *PR_AddToGroup()* function.
PRNAMETOID	504	Opcode for the *PR_NameToID()* function.
PRIDTONAME	505	Opcode for the *PR_IDToName()* function.
PRDELETE	506	Opcode for the *PR_Delete()* function.
PRREMOVEFROMGROUP	507	Opcode for the *PR_RemoveFromGroup()* function.
PRGETCPS	508	Opcode for the *PR_GetCPS()* function.
PRNEWENTRY	509	Opcode for the *PR_NewEntry()* function.
PRLISTMAX	510	Opcode for the *PR_ListMax()* function.
PRSETMAX	511	Opcode for the *PR_SetMax()* function.
PRLISTENTRY	512	Opcode for the *PR_ListEntry()* function.
PRCHANGEENTRY	513	Opcode for the *PR_ChangeEntry()* function.
PRLISTELEMENTS	514	Opcode for the *PR_ListElements()* function.
PRISAMEMBEROF	515	Opcode for the *PR_IsAMemberOf()* .
PRSETFIELDSENTRY	516	Opcode for the *PR_SetFieldsEntry()* function.
PRLISTOWNED	517	Opcode for the *PR_ListOwned()* function.
LOWEST	500	Lowest-numbered *Protection Server* opcode.
HIGHEST	517	Highest-numbered *Protection Server* opcode.

3.2.2 Size and Configuration Quantities

This section presents some miscellaneous values defined within the *Protection Server* which have to do with sizes for various structures as well as general configuration facts. For example, maximum sizes for names used within the *Protection Server* as well as for lists generated and passed to clients are specified. Also, hash table sizes and ID array allocations within the PDB records are nailed down. Such configuration information as the currently-supported PDB database version number and the *Rx* service identifier advertised by the *Protection Server* for handling of its calls appears here.

Name	*Value*	*Description*
`PR_MAXNAMELEN`	64	Longest permissible AFS user or group name, including the trailing null.
`PR_MAXGROUPS`	5,000	Maximum number of group IDs which can fit into a `prlist` variable-length array (See Section 3.3.9).
`PR_MAXLIST`	5,000	Maximum number of string names or numerical IDs which can fit into a `namelist` or `idlist` variable-length array (see Sections 3.3.7 and 3.3.8).
`PRSIZE`	10	Maximum number of IDs kept in a `struct prentry` in the PDB before having to resort to a continuation block.
`COSIZE`	39	Maximum number of IDs kept in a PDB continuation block.
`PRSRV`	73	*Rx* service identifier for the *Protection Server*.
`ENTRYSIZE`	192	Size in bytes of a PDB entry.
`HASHSIZE`	8,191	Size of the internal `nameHash` and `idHash` *Protection Server* hash lookup tables.
`PRDBVERSION`	0	PDB version number.

3.2.3 Mask Bits for *PR_SetFieldsEntry()*

The *PR_SetFieldsEntry()* interface call allows various fields of a PDB entry to be altered, as described in detail in Section 3.6.8. One of the arguments to that function is a mask whose bit values are used to select which fields are to be assigned. This section presents the valid sets of bits that can be passed in this mask parameter.

Name	*Value*	*Description*
`PR_SF_NGROUPS`	(1<<31)	Set the field limiting group creation.
`PR_SF_NUSERS`	(1<<30)	Set the field limiting creation of foreign users.
`PR_SF_ALLBITS`	0xff	Set all access (privacy) bits.

3.2.4 Reserved IDs

Certain AFS user and group IDs are reserved, and have special meaning to the system. This section reveals the set of reserved IDs and explains their significance.

Name	*Value*	*Description*
SYSADMINID	-204	This ID is used for the special `system:administrators` group, namely those individuals declared to be capable of performing certain administrative AFS operations.
ANYUSERID	-101	This ID is used for the special `system:anyuser` group, namely all AFS users, regardless of their authentication state.
AUTHUSERID	-102	This ID is used for the special `system:authuser` group, namely all AFS users which hold valid Kerberos tickets within the local cell.
ANONYMOUSID	32,766	This ID is used for the special AFS user which corresponds to someone who has not generated tickets within the local cell.
PRBADID	0x80000000	This ID is reserved to indicate an AFS ID which is guaranteed to be "bad" (i.e., unused), and is used to mark empty slots in a PDB entry.

3.2.5 Bits for `struct prentry flags` Field

A flag field is maintained in the structure which defines the format of PDB records. These flag bits carry information as to the nature of the entry and whether such things as group creation quotas and special access checking is being obeyed. Specifically, the least-significant 6 bits identify the type of the PDB entry, and the next two bits determine whether enforcement of the above two policies is in effect. This section lays out the bits in this field and reports on their individual meanings.

Name	*Value*	*Description*
PRTYPE	0x3f	Bit mask isolating the type bits for a PDB entry. Only one of these bits should be set.
PRFREE	1	The PDB entry is in the free list.
PRGRP	2	The PDB entry is a group entry.
PRCONT	4	The PDB entry is formatted as a continuation block.
PRCELL	8	The PDB entry is a cell entry *(not used)*.
PRFOREIGN	16	The PDB entry is for a foreign user *(not used)*.
PRINST	32	The PDB entry is for a sub/super instance of an identifier *(not used)*.
PRUSER	0	If all type bits (the low-order 6 bits) are 0, then the entry is a user entry.
PRACCESS	(1<<6)	Access checking is enabled or this entry.
PRQUOTA	(1<<7)	Group creation quota checking is enabled for this entry.

3.2.6 Privacy Bits

As presented in some detail in Section 2.2.2, the ability to list information regarding the individual PDB entries is controlled by a set of bits in the entry itself. This section covers the organization of these privacy bits. The bit aray is stored in the upper 16 bits of the `flags` field of the `struct prentry` defining the layout of PDB entries. Note, however, that the `flags` argument to the *PR_SetFieldsEntry()* RPC interface routine which allows access to these bits expects these bits in the lower 16 bits of the word. For convenience, the `PRIVATE_SHIFT` constant is defined so that programmers can shift the bit fields to their proper locations. There are a total of 8 bit positions defined. Two of the five rights (`OWNERED`, `REMOVE`) only require one bit to encode. The others (`STATUS`, `MEMBERSHIP`, and `ADD`) require two bits to represent, as they have more "levels" at which they may be set.

Name	*Value*	*Description*
PRIVATE_SHIFT	16	Constant used to shift privacy bits into proper position in the `flags` field.
PRP_REMOVE_MEM	(0x01 << PRIVATE_SHIFT)	Allows members of the given group to remove users from its membership.
PRP_ADD_MEM	(0x02 << PRIVATE_SHIFT)	Allows members of a group to delete users from its membership.
PRP_ADD_ANY	(0x04 << PRIVATE_SHIFT)	Allows anyone to add users to a group's membership.
PRP_MEMBER_MEM	(0x08 << PRIVATE_SHIFT)	Allows members of a group to list its membership set
PRP_MEMBER_ANY	(0x10 << PRIVATE_SHIFT)	Allows anyone to list the membership set of a group, or list the set of groups to which a given individual belongs.
PRP_OWNED_ANY	(0x20 << PRIVATE_SHIFT)	Allows the viewing of the list of groups owned by the given entry.
PRP_STATUS_MEM	(0x40 << PRIVATE_SHIFT)	Allows members of a group to get status on the given entry.
PRP_STATUS_ANY	(0x80 << PRIVATE_SHIFT)	Allows anyone to get status on a given entry.

In addition to those constant definitions defined in the above table, there are two other constants that are useful in this realm. The `PRP_USER_DEFAULT` constant, set to `(PRP_STATUS_ANY)`, defines the default access allowed to a user entry in the PDB when the `PRACCESS` bit is turned off. It states that any user may obtain status information on the entry, but only the given user and members of the `system:administrators` group can list the groups they own and the groups of which they are members. This corresponds to an interface specification of `S-----`. Similarly, the `PRP_GROUP_DEFAULT` constant, defined as `(PRP_STATUS_ANY | PRP_MEMBER_ANY)`, dictates the standard access allowed for a group entry in the PDB. As set, any user may obtain status information on the entry and list the group's members. This corresponds to an interface specification of `S-M--`.

3.2.7 Miscellaneous

The following constant definitions don't fall easily into the above categrories, so are collected here.

Name	*Value*	*Description*
`PRSUCCESS`	0	Defined to match the "successful" return value from *Protection Server* routines.
`PR_REMEMBER_TIMES`	1	If non-zero, the *Protection Server* will record the clock times when a PDB entry was created, had an ID added to it, had an ID removed from it, and the time of the last change of any type. This information is held in the 5 `reserved` longwords in `struct prentry`.

3.3 Structures and Typedefs

This section describes the major exported *Protection Server* data structures of interest to application programmers, along with the typedefs based upon those structures. Please note that some of the typedefs included here were made for the RPC stub generator, *rxgen*. *Rxgen* takes a specification of a server's procedure call interface as input, and generates the proper server-side and client-side C stub code, along with any necessary C include files. These typedefs included for *rxgen* handling may be identified by the fact that they contain angle brackets as part of the definition.

3.3.1 `struct prheader`

This structure defines the format of the PDB header record. It is reponsible for recording information about the PDB as a whole. Examples of the knowledge stored here are the PDB's version number, the address of the beginning of the free list, the number of users and groups in the system, the largest user and group IDs allocated so far, and the hash tables used for looking up entries by name and by ID. There are also some fields included here for features which have not been implemented, such as having foreign users in the PDB (users registered with another cell) and having superset or subset instances.

The PDB's header region is not directly visible via the *Protection Server* interface, but such interface calls as *PR_ListMax()*, *PR_SetMax()*, and even *PR_ListOwned()* when it's used to deliver the orphan list report on the contents of some of the header's fields,

Fields

long version - The PDB's database version number

long headerSize - The size of this structure in bytes.

long freePtr - Pointer to the first free PDB entry, namely the first entry in the free list.

long eofPtr - Pointer to the high-water mark in the PDB replicated file, namely the first completely unused byte in the file and *not* any part of the free list.

long maxGroup - The "largest" (most negative) group ID allocated to date.

long maxID - The "largest" (most positive) user id allocated to date.

long maxForeign - The largest foreign ID allocated to date *(not implemented)*.

long maxInst - The largest subset or superset ID allocated to date *(not implemented)*.

long orphan - The list of groups owned by deleted users.

long usercount - The total number of users for which there are PDB entries in the system.

long groupcount - The total number of groups for which there are PDB entries in the system.

long foreigncount - The number of registered foreign users *(not implemented)*.

long instcount - The number of subset and superset users in the system *(not implemented)*.

long reserved[] - Five longword values reserved for future use.

long nameHash[] - The name-based hash lookup table, an array of `HASHSIZE` (8,191) PDB entry pointers.

long idHash[HASHSIZE] - The ID-based hash lookup table, is implemented as an array of `HASHSIZE` (8,191) PDB entry pointers.

3.3.2 `struct prentry`

This structure represents the format of the "main" PDB entry for an AFS user or group. Along with the other information concerning the given user or group, it holds up to `PRSIZE` (10) IDs. If the entry is for a user, then these IDs list the set of groups to which the individual belongs. If the entry is for a group, then the IDs represent the individuals that belong to the group. If there are more than `PRSIZE` IDs to keep track of for the entry, then it must use one or more *continuation blocks*, as defined by the `struct contentry` structure described in Section 3.3.3. Please note that "main" PDB entries and continuation entries are both the same size (`ENTRYSIZE` (192) bytes), but differ in format.

The reader should note that the `struct prdebugentry` defined in Section 3.3.4 below is identical to this structure. The `prdebugentry` version, though, is the externally-visible version, and is only used during debugging via the *PR_DumpEntry()* interface routine.

Fields

long flags - These are bits having to do with the entry type, the operations enabled on this entry, and the access checks that will be performed. Please refer to Sections 3.2.5 and 3.2.6 for the legal values for this field.

long id - The associated user or group ID.

long cellid - The entry's cell identifier *(not used)*.

long next - A pointer to the next PDB block for the same entry, or the next block on the free list if the entry currently resides there.

reserved[] - This field consists of 5 (semi-)reserved unsigned longwords, put to use in different ways selectable at *Protection Server* compile-time. If the `PR_REMEMBER_TIMES` constant is defined to be non-zero, then the first four longwords are interpreted as fields whose names are `createTime`, `addTime`, `removeTime`, and `changeTime`. They refer respectively to the time of creation of the entry, the last time an ID was added to it, the last time an ID was removed from it, and the last time a change of any other kind was made. In this case, the fifth longword in the series is referred to as `reserved[1]`. Should the `PR_REMEMBER_TIMES` be set to zero at compilation time, the code will refer to these five longwords as `reserved[5]`.

long entries[] - An array of AFS IDs associated with the given entry. If the entry represents a group, then each ID refers to an individual member of the group. If the entry represents an individual user, then each ID refers to a group to which that user belongs. This array holds a maximum of `PRSIZE` (10) entries.

long nextID - Next pointer in the by-ID PDB hash table.

long nextName - Next pointer in the by-name PDB hash table.

long owner - The AFS ID of this entry's owner.

long creator - The AFS ID of the party which created this entry.

long ngroups - The number of groups this ID has created. If this is a group entry, `ngroups` is always 0.

long nusers - The number of foreign user entries this ID has created *(not used)*.

long count - The number of members/groups for this group/user.

long instance - The number of superset/subset instances for this user *(not used)*.

long owned - A pointer to the head of a chain of groups owned by this entry. Note that both users and groups can own groups.

long nextOwned - The chain of groups for the owner of this entry.

long parent - Pointer to the entry's superset instance *(not used)*.

long sibling - Pointer to this entry's sibling instance *(not used)*.

long child - Pointer to this entry's first child *(not used)*.

char name[] - The printable name of the associated user or group. It has at most `PR_MAXNAMELEN` (64) characters in it, including the trailing null.

3.3.3 `struct contentry`

This structure represents the format of a "continuation" PDB entry for an AFS user or group. Along with a subset of the information held in the "main" entry (of type `struct prentry`), it maintains up to `COSIZE` (39) additional IDs associate with the extended entry. If the entry is for a user, then these IDs list the set of groups to which the individual belongs. If the entry is for a group, then the IDs represent the individuals that belong to the group. As pointed out above, "main" PDB entries and continuation entries are both the same size (`ENTRYSIZE` (192) bytes), and simply differ in format.

Fields

long flags - These are bits having to do with the entry type, the operations enabled on this entry, and the access checks that will be performed. Please refer to Sections 3.2.5 and 3.2.6 for the legal values for this field.

long id - The associated user or group ID.

long cellid - The entry's cell identifier *(not used)*.

long next - A pointer to the next PDB block for the same entry, or the next block on the free list if the entry currently resides there.

long reserved[] - The set of 5 reserved words which may or may not hold certain time values related to the entry. If the `PR_REMEMBER_TIMES` constant is defined to be non-zero, then the first four longwords are interpreted as fields specifying the time of creation of the entry, the last time an ID was added to it, the last time an ID was removed from it, and the last time a change of any kind was made. The fifth longword in the series is still unused and reserved for future assignment.

long entries[] - An array of additional AFS IDs associated with the given entry's main ID. If the entry represents a group, then each ID refers to an individual member of the group. If the entry represents an individual user, then each ID refers to a group to which that user belongs. This array holds a maximum of `COSIZE` (39) entries.

3.3.4 `struct prdebugentry`

This structure is identical to the basic PDB entry structure defined as `struct prentry` in Section 3.3.2, except for a minor spelling modification of one field (`nextName` in `struct prentry`, `nextname` in this structure). This construct is used exclusively by the *PR_DumpEntry()* interface function. For the reader's convenience, the description of its fields is repeated here.

Fields

long flags - These are bits having to do with the entry type, the operations enabled on this entry, and the access checks that will be performed. Please refer to Sections 3.2.5 and 3.2.6 for the legal values for this field.

long id - The associated user or group ID.

long cellid - The entry's cell identifier *(not used)*.

long next - A pointer to the next PDB block for the same entry, or the next block on the free list if the entry currently resides there.

reserved[] - This field consists of 5 (semi-)reserved unsigned longwords, put to use in different ways selectable at *Protection Server* compile-time. If the `PR_REMEMBER_TIMES` constant is defined to be non-zero, then the first four longwords are interpreted as fields whose names are `createTime`, `addTime`, `removeTime`, and `changeTime`. They refer respectively to the time of creation of the entry, the last time an ID was added to it, the last time an ID was removed from it, and the last time a change of any kind was made. In this case, the fifth longword in the series is referred to as `reserved[1]`. Should the `PR_REMEMBER_TIMES` be set to zero at compilation time, the code will refer to these five longwords as `reserved[5]`.

long entries[] - An array of AFS IDs associated with the given entry. If the entry represents a group, then each ID refers to an individual member of the group. If the entry represents an individual user, then each ID refers to a group to which that user belongs. This array holds a maximum of `PRSIZE` (10) entries.

long nextID - Next pointer in the by-ID PDB hash table.

long nextname - Next pointer in the by-name PDB hash table.

long owner - The AFS ID of this entry's owner.

long creator - The AFS ID of the party which created this entry.

long ngroups - The number of groups this ID has created. If this is a group entry, `ngroups` is always 0.

long nusers - The number of foreign user entries this ID has created *(not used)*.

long count - The number of members/groups for this group/user.

long instance - The number of superset/subset instances for this user *(not used)*.

long owned - The chain of groups owned by this entry.

long nextOwned - The chain of groups for the owner of this entry.

long parent - Pointer to the entry's superset instance *(not used)*.

long sibling - Pointer to this entry's sibling instance *(not used)*.

long child - Pointer to this entry's first child *(not used)*.

char name[] - The printable name of the associated user or group. It has at most `PR_MAXNAMELEN` (64) characters in it, including the trailing null.

3.3.5 `struct prcheckentry`

This structure is a subset of the fields defined within the `struct prentry` described in Section 3.3.2. It is used to communicate the "user-visible" fields of a PDB entry to the average inquirer, and appears as an argument to the *PR_ListEntry()* RPC interface call, described in Section 3.6.5.

Fields

long flags - These are bits having to do with the entry type, the operations enabled on this entry, and the access checks that will be performed. Please refer to Sections 3.2.5 and 3.2.6 for the legal values for this field.

long id - The associated user or group ID.

long owner - The AFS ID of this entry's owner.

long creator - The AFS ID of the party which created this entry.

long ngroups - The number of groups this ID has created. If this is a group entry, `ngroups` is always 0.

long nusers - The number of foreign user entries this ID has created *(not used)*.

count - The number of members/groups for this group/user.

reserved[] - The set of 5 reserved words which may or may not hold certain time values related to the entry. If the `PR_REMEMBER_TIMES` constant is defined to be non-zero, then the first four longwords are interpreted as fields specifying the time of creation of the entry, the last time an ID was added to it, the last time an ID was removed from it, and the last time a change of any kind was made. The fifth longword in the series is still unused and reserved for future assignment.

name[] - The printable name of the associated user or group. It has at most `PR_MAXNAMELEN` (64) characters in it, including the trailing null.

3.3.6 typedef prname

```
typedef char prname[PR_MAXNAMELEN];
```

This typedef is used to define a variable which houses the printable name of an AFS principal. Such names may be up to `PR_MAXNAMELEN` (64) characters long, including the trailing null. In turn, `prname` is used by the `namelist` typedef to create an array of these printable user names (see Section 3.3.7).

3.3.7 tyepdef namelist

```
typedef prname namelist<PR_MAXLIST>;
```

This typedef is used by *rxgen* to define a C structure providing a variable-length array of printable AFS user names. It may carry up to `PR_MAXLIST` (5,000) of these name entries, and contains the following fields:

Fields

`namelist_len` - The number of name entries contained within the data region pointed to by the next field.

`namelist_val` - A pointer to a sequence of `namelist_len` AFS name entries.

The `namelist` typedef is used in the *PR_NameToID()* and *PR_IDToName()* RPC interface calls (see Sections 3.6.1 and 3.6.2 respectively).

3.3.8 typedef idlist

```
typedef long idlist<PR_MAXLIST>;
```

This typedef is used by *rxgen* to define a C structure providing a variable-length array of numerical AFS user identifiers. It may carry up to `PR_MAXLIST` (5,000) of these IDs, and contains the following fields:

Fields

`idlist_len` - The number of longword IDs contained within the data region pointed to by the next field.

`idlist_val` - A pointer to a sequence of `idlist_len` numerical AFS user IDs.

The `idlist` typedef is used in the *PR_NameToID()* and *PR_IDToName()* RPC interface calls (see Sections 3.6.1 and 3.6.2 respectively).

3.3.9 typedef prlist

```
typedef long prlist<PR_MAXGROUPS>;
```

This typedef is used by *rxgen* to define a C structure providing a variable-length array of AFS IDs. This structure is used as an argument to the *PR_GetCPS()*, *PR_ListElements()*, and *PR_ListOwned()* (described in Sections 3.6.16, 3.6.15, and 3.6.17 respectively). It may carry up to `PR_MAXGROUPS` (5,000) of these IDs, and sports the following fields:

Fields

`prlist_len` - The number of longword AFS IDs contained within the data pointed to by the next field.

`prlist_val` - A pointer to a sequence of `prlist_len` IDs.

3.4 Error Codes

This section covers the set of error codes exported by the *Protection Server*, displaying the printable phrases with which they are associated.

Name	*Value*	*Description*
PREXIST	(267264L)	Entry for name already exists.
PRIDEXIST	(267265L)	Entry for id already exists.
PRNOIDS	(267266L)	Couldn't allocate an id for this entry.
PRDBFAIL	(267267L)	Couldn't read/write the database.
PRNOENT	(267268L)	User or group doesn't exist.
PRPERM	(267269L)	Permission denied.
PRNOTGROUP	(267270L)	No group specified.
PRNOTUSER	(267271L)	No user specified.
PRBADNAM	(267272L)	Badly formed name (group prefix doesn't match owner?).
PRBADARG	(267273L)	argument illegal or out of range.
PRNOMORE	(267274L)	may not create more groups.
PRDBBAD	(267275L)	database needs rebuilding.
PRGROUPEMPTY	(267276L)	can't make owner an empty group.
PRINCONSISTENT	(267277L)	database is inconsistent.
PRDBADDR	(267278L)	bad database address.
PRTOOMANY	(267279L)	too many elements in group.

3.5 Macros

The *Protection Server* defines a pair of macros which are externally visible via the *ptserver.h* file. They are both used to operate on fields within the PDB header structure. These two macros are only invoked internally, and are not usable from a normal AFS application.

3.5.1 *set_header_word()*

```
#define set_header_word(tt, field, value) \
    pr_Write((tt), 0, ((char *)&(cheader.field) - (char *)&cheader),   \
        ((cheader.field = (value)), (char *)&(cheader.field)),         \
        sizeof(long))
```

This macro is used to transactionally assign a field within the PDB header structure. The first argument, `tt`, is a pointer of type `(struct ubik_trans *)`, and communicates the *ubik* transaction under which this update is taking place. The next argument, `field`, identifies the name of the field within the PDB header to be changed. Finally, the `value` parameter represents the new value to be assigned to `field`. Only PDB header fields of type `long` may be assigned in this fashion.

3.5.2 *inc_header_word()*

```
#define inc_header_word(tt, field, inc) \
    pr_Write((tt), 0, ((char *)&(cheader.field) - (char *)&cheader), \
        ((cheader.field = (htonl(ntohl(cheader.field)+(inc)))),      \
         (char *)&(cheader.field)),                                  \
        sizeof(long))
```

This macro is used to transactionally increment a field within the PDB header structure. The first argument, `tt`, is a pointer of type `(struct ubik_trans *)`, and communicates the *ubik* transaction under which this update is taking place. The next argument, `field`, identifies the name of the field within the PDB header to be changed. Finally, the `inc` parameter specifies the amount the chosen field should be incremented. Only PDB header fields of type `long` may be incremented in this fashion.

3.6 Functions

This section covers the *Protection Server* RPC interface routines. They are generated from the *ptint.xg Rxgen* file. The following is a summary fo the interface functions and their purpose:

Fcn Name	**Description**
PR_NameToID()	Perform name-to-ID translation(s).
PR_IDToName()	Perform ID-to-name translation(s).
PR_NewEntry()	Create a PDB entry for the given name.
PR_INewEntry()	Create a PDB entry for the given name *and* ID.
PR_ListEntry()	Get the contents of a PDB entry based on its ID.
PR_DumpEntry()	Get the contents of a PDB entry based on its offset (for debugging).
PR_ChangeEntry()	Change an existing PDB entry's ID, name, and/or owner.
PR_SetFieldsEntry()	Change miscellaneous fields in an existing PDB entry.
PR_Delete()	Delete an existing PDB entry.
PR_WhereIsIt()	Get the PDB byte offset of the entry for a given ID. This is intended for debugging purposes.
PR_AddToGroup()	Add a user to a group.
PR_RemoveFromGroup()	Remove a user from a chosen group.
PR_ListMax()	Get the largest allocated user and group ID.
PR_SetMax()	Set the largest allocated user and group ID.
PR_ListElements()	List all IDs associated with a user or group.
PR_GetCPS()	Get the CPS for the given ID.
PR_ListOwned()	List all IDs owned by the given ID.
PR_IsAMemberOf()	Is a given user ID a member of a specified group?.

At a higher level, these functions may be seen as belonging to four basic classes:

- Name to ID mapping, and vice-versa.
- Creation, inspection, modification, and deletion of PDB entries.
- Inquiries on selected PDB header fields.
- Ownership/membership queries and modifications.

3.6.1 PR_NameToID — Perform name-to-ID translation(s)

```
int PR_NameToID(IN struct rx_connection *z_conn,
                IN namelist *nlist,
                OUT idlist *ilist)
```

Description

This routine takes as an input parameter a variable-length array of AFS user names, `nlist`. Its job is to then translate each of the `nlist->namelist_len` names into the appropriate numerical AFS user ID. Values greater than or equal to zero and less than or equal to `PR_MAXLIST` (5,000) are legal values for `nlist->namelist_len`. The output parameter, `ilist`, is used to store the list of IDs matching the given names. Exactly `nlist->namelist_len` entries are allocated for the variable-length `ilist` output array.

Entries in the given `nlist` for which the *Protection Server* does not have a name-to-ID mapping are translated to `ANONYMOUSID`, the anonymous user ID. Since this is a potentially long-lived operation, a lightweight *Protection Server* thread executing this call offers itself for rescheduling every 50 translations. This threading activity is completely invisible to the caller.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

`PRTOOMANY` An illegal value was found in the `namelist_len` field within the `nlist` argument, either less than or equal to zero or more than `PR_MAXLIST` (5,000).

`PRDBBAD` An error was found relating to the PDB.

3.6.2 PR_IDToName — Perform ID-to-name translation(s)

```
int PR_IDToName(IN struct rx_connection *z_conn,
                IN idlist *ilist,
                OUT namelist *nlist)
```

Description

This routine takes as an input parameter a variable-length array of numerical AFS user IDs, `ilist`. Its job is to then translate each of the `ilist->idlist_len` IDs into the appropriate printable AFS user names. Values greater than or equal to zero and less than or equal to `PR_MAXLIST` (5,000) are legal values for `ilist->idlist_len`. The output parameter, `nlist`, is used to store the list of names matching the given IDs. Exactly `ilist->idlist_len` entries are allocated for the variable-length `nlist` output array.

Entries in the given `ilist` for which the *Protection Server* does not have an ID-to-name mapping are translated to the string representation of the numerical ID, in base 10. Since this is a potentially long-lived operation, a lightweight *Protection Server* thread executing this call offers itself for rescheduling every 50 translations. This threading activity is completely invisible to the caller.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

- `PRTOOMANY` An illegal value was found in the `ilist_len` field within the `ilist` argument, either less than or equal to zero or more than `PR_MAXLIST` (5,000).
- `PRDBBAD` An error was found relating to the PDB.

3.6.3 PR_NewEntry — Create a PDB entry for the given name

```
int PR_NewEntry(IN struct rx_connection *z_conn,
                IN char *name,
                IN long flag,
                IN long oid,
                OUT long *id)
```

Description

Create an entry in the PDB for a user or group with the given `name`, and whose owner's ID is `oid`. The `name` parameter must have no more than `PR_MAXNAMELEN` (64) characters, including the trailing null. The `flags` argument is used to communicate whether the entry is for a user or group. It may legally take on any of the values `PRGRP`, `PRUSER`, or `PRFOREIGN` (although foreign IDs are not implemented), as defined in Section 3.2.5. If successful, the user or group ID allocated for this new entry is returned in `id`.

Creation of a group entry by an unauthenticated caller is allowed only if the owner (`oid`) is specified as a non-zero value, or if the *Protection Server* is running in an unauthenticated mode. Creation of a user entry is allowed only if the caller is a member of the `system:administrators` group, or if the *Protection Server* is running in an unauthenticated mode. Members of the `system:administrators` group may create an unlimited number of groups, while other users are restricted by a group quota, as described in Section 2.1.1.

The underlying *ubik*-based PDB is manipulated under a write transaction, and is write-locked for the duration of the operation.

Error Codes

`PRPERM` The caller's identity could not be established, or is not authorized to perform the operation.

`PRBADNAM` An Illegal name was provided for the new entry.

`PREXIST` The named user or group already exists.

`PRDBFAIL` Space for a new PDB entry could not be allocated, or the entry could not be successfully filled in.

`PRNOMORE` The caller's creation quota has been exceeded.

`PRDBBAD` An error was found relating to the PDB.

3.6.4 PR_INewEntry — Create a PDB entry for the given name *and* ID

```
int PR_INewEntry(IN struct rx_connection *z_conn,
                 IN char *name,
                 IN long id,
                 IN long oid)
```

Description

This routine is similar to the *PR_NewEntry()* function described in Section 3.6.3 in that it creates a new user or group entry in the PDB. However, *PR_INewEntry()* is used when the numerical AFS ID *must* be set to a given value, namely `id`. Thus, there are no output parameters here. This function is not normally used in every-day operation. Rather, it is particularly useful for conversion operations, where AFS identities are being created in the PDB based on an existing set of UNIX ids.

Please refer to the description of the *PR_NewEntry()* interface function for more details.

Error Codes

`PRPERM` The caller's identity could not be established, or is not authorized to perform the operation.

`PRBADNAM` An illegal name was provided for the new entry.

`PREXIST` The named user or group already exists.

`PRDBFAIL` Space for a new PDB entry could not be allocated, or the entry could not be successfully filled in.

`PRNOMORE` The caller's creation quota has been exceeded.

`PRDBBAD` An error was found relating to the PDB.

3.6.5 PR_ListEntry — Get the contents of a PDB entry based on its ID

```
int PR_ListEntry(IN struct rx_connection *z_conn,
                 IN long id,
                 OUT struct prcheckentry *entry)
```

Description

This function looks up the PDB entry associated with the numerical identifier `id` and writes the associated information into the `entry` parameter. As described in Section 3.3.5, a variable of type `struct prcheckentry` houses the entry's externally-visible fields for the casual inquirer.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

- `PRPERM` The caller's identity could not be established, or is not authorized to perform the operation.
- `PRNOENT` A PDB entry matching `id` was not found.
- `PRDBBAD` An error was found relating to the PDB.

3.6.6 PR_DumpEntry — Get the contents of a PDB entry based on its offset (for debugging)

```
int PR_DumpEntry(IN struct rx_connection *z_conn,
                 IN long pos,
                 OUT struct prdebugentry *entry)
```

Description

This routine returns the contents of the PDB entry located at byte offset `pos` within the replicated database, and is intended to be used only for debugging. The text of the entry is deposited into the buffer pointed to by `entry`. Note that the `struct debugentry` used for the `entry` argument is identical to the `struct prentry` which is used directory by the PDB (See Sections 3.3.2 and 3.3.4). Since the given offset may point in fact to a continuation entry instead of a main PDB entry, the caller must be prepared to interpret the contents of the `entry` argument, of type `struct prdebugentry`, as an object of type `struct contentry`, as described in Section 3.3.3.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

`PRDBBAD` An error was found relating to the PDB.

3.6.7 PR_ChangeEntry — Change an existing PDB entry's ID, name, and/or owner

```
int PR_ChangeEntry(IN struct rx_connection *z_conn,
                   IN long id,
                   IN char *name,
                   IN long oid,
                   IN long newid)
```

Description

This routine allows the caller to change one or more aspects of the user or group entry in the PDB whose numerical ID is `id`. An attempt to change the entry for any of the reserved AFS IDs (`ANYUSERID`, `AUTHUSERID`, `ANONYMOUSID`, or `SYSADMINID`) will be rejected. The new name, if any, to assign to the entry is held in `name`. If no name change is to be made, then the `name` argument may be set to the null string. In any case, the specified name may be no longer than `PR_MAXNAMELEN` (64) characters long, including the trailing null. Similarly, the numerical ID of the entry's (possibly new) owner is listed in the `oid` parameter. If no owner change is desired, `oid` is set to zero. Finally, a new numerical ID may be specified for the given entry in `newid`. If this is set to zero, then the entry's ID will not be altered.

The caller will be allowed to make changes to a PDB entry under the following general conditions:

- They are the entry's owner.
- They are a member of the `system:administrators` group.
- They are a member of the group which is the owner of the given entry.
- The *Protection Server* is running in its unauthenticated mode.

There are other specific conditions governing altering specific fields within the entry:

- The numerical ID may only be changed if the caller is a member of the built-in AFS administration group, `system:administrators`.

- The name and owner ID must be consistent with group naming rules following the requested changes unless the caller is a system administrator. Otherwise, `PRBADNAM` is returned.
- The entry's owner may only be changed if the entry is for a group, not for an individual user.

The underlying *ubik*-based PDB is manipulated under a write transaction, and is write-locked for the duration of the operation.

Error Codes

`PRPERM` The caller's identity could not be established, or is not authorized to perform the operation, or an attempt was made to change a group entry to a user entry (or vice versa), or an attempt was made to change a user entry's owner, or an attempt was made to change a reserved ID.

`PRNOENT` A PDB entry matching `id` was not found.

`PRIDEXIST` The new ID specified by `newid` is already in use.

`PRBADNAM` An illegal `name` was provided for the new entry.

`PRDBFAIL` Reading or writing the given PDB entry failed.

`PRDBBAD` An error was found relating to the PDB.

3.6.8 PR_SetFieldsEntry — Change miscellaneous fields in an existing PDB entry

```
int PR_SetFieldsEntry(IN struct rx_connection *z_conn,
                      IN long id,
                      IN long mask,
                      IN long flags,
                      IN long ngroups,
                      IN long nusers,
                      IN long spare1,
                      IN long spare2)
```

Description

This function allows a caller to change one or more fields within the PDB entry matching the given user `id`. The `mask` argument is used to select which fields are to be changed. Legal values for this `mask` parameter are provided in Section 3.2.3. Depending on what has been selected, the values placed in the `flags` (setting the flag bits), `ngroups` (setting the group creation limit), and/or `nusers` (setting the foreign user limit) are inserted into the PDB entry. The remaining two arguments, `spare1` and `spare2`, are not referenced, and are reserved for future expansion.

Attempting to alter the Protection Database entries for certain AFS IDs, namely ANYUSERID, AUTHUSERID, ANONYMOUSID, and SYSADMIN are rejected.

The underlying *ubik*-based PDB is manipulated under a write transaction, and is write-locked for the duration of the operation.

Error Codes

PRPERM The caller's identity could not be established, or is not authorized to perform the operation.

PRBADARG One or more of the values provided as arguments are illegal.

PRDBBAD An error was found relating to the PDB.

3.6.9 PR_Delete — Delete an existing PDB entry

```
int PR_Delete(IN struct rx_connection *z_conn,
              IN long id)
```

Description

This operation is used to delete the PDB entry for the user or group whose ID is `id`. The caller must be either the entry's owner, a member of the `system:administrators` group, or a member of the group listed as the entry's owner for this function to succeed. Alternatively, the *Protection Server* will allow the operation if it is running in its unauthenticated mode.

Since this is a potentially long-lived operation, it may be split up into several *ubik* write transactions, with each one holding a write lock on the PDB. The processing of each continuation block (if any) for the entry is handled in a separate transaction. Similarly, a new transaction is generated for every 50 entries handled in the entry's `owned` list. In addition, the lightweight *Protection Server* thread executing this call offers itself for rescheduling after handling each continuation block and after handling each group of 4 items in the `owned` list.

Error Codes

`PRDBBAD` An error was found relating to the PDB.

`PRPERM` The caller's identity could not be established, or is not authorized to perform the operation, or tried to delete a reserved AFS ID.

`PRNOENT` A PDB entry matching `id` was not found.

`PRDBFAIL` The PDB could not be read from or written to successfully.

3.6.10 PR_WhereIsIt — Get the PDB byte offset of the entry for a given ID

```
int PR_WhereIsIt(IN struct rx_connection *z_conn,
                 IN long id,
                 OUT long *ps);
```

Description

This interface function will look up the PDB entry associated with the given numerical AFS `id` and return the byte offset into the replicated PDB file where the entry is located. This may be used in conjunction with the *PR_DumpEntry()* call to examine the contents of the PDB at the given offset.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

`PRNOENT` A PDB entry matching `id` was not found.

`PRDBBAD` An error was found relating to the PDB.

3.6.11 PR_AddToGroup — Add a user to the chosen group

```
int PR_AddToGroup(IN struct rx_connection *z_conn,
                  IN long uid,
                  IN long gid)
```

Description

This routine is used to add the given AFS `uid` to the group whose numerical identifier is `gid`. Note that group identifiers may not appear within a group.

The underlying *ubik*-based PDB is manipulated under a write transaction, and is write-locked for the duration of the operation.

Error Codes

- `PRPERM` The caller's identity could not be established, or is not authorized to perform the operation, or tried to add a member to one of the two special AFS built-in groups, `system:anyuser` and `system:authuser`, or tried to add `ANONYMOUSID` to the group.
- `PRNOENT` A PDB entry matching `uid` or `gid` was not found.
- `PRNOTUSER` An attempt was made to add a group identifier to a group.
- `PRNOTGROUP` The specified `gid` is not a group.
- `PRDBBAD` An error was found relating to the PDB.

3.6.12 PR_RemoveFromGroup — Remove a user from the chosen group

```
int PR_RemoveFromGroup(IN struct rx_connection *z_conn,
                       IN long id,
                       IN long gid)
```

Description

This function allows the caller to remove the given AFS numerical `id` from the group identified by `gid`.

The underlying *ubik*-based PDB is manipulated under a write transaction, and is write-locked for the duration of the operation.

Error Codes

`PRPERM` The caller's identity could not be established, or is not authorized to perform the operation, or tried to remove a member to one of the two special AFS built-in groups, `system:anyuser` and `system:authuser`, or tried to add `ANONYMOUSID` to the group.

`PRNOENT` A PDB entry matching `id` or `gid` was not found.

`PRNOTUSER` An attempt was made to remove a group identifier from the given group.

`PRNOTGROUP` An attempt was modify a PDB entry that does not describe a group.

`PRDBBAD` An error was found relating to the PDB.

3.6.13 PR_ListMax — Get the largest allocated user/group ID

```
int PR_ListMax(IN struct rx_connection *z_conn,
               OUT long *uid,
               OUT long *gid)
```

Description

This *Protection Server* interface routine places into its `uid` and `gid` arguments the largest allocated AFS numerical user and group IDs respectively.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

`PRDBBAD` An error was found relating to the PDB.

3.6.14 PR_SetMax — Set the largest allocated user/group ID

```
int PR_SetMax(IN struct rx_connection *z_conn,
              IN long id,
              IN long gflag)
```

Description

This routine is used to set the value of either the highest-allocated AFS user or group to `id`. The `gflag` argument is set to a non-zero value if the given `id` is for a group, and to zero for an individual user. After successful completion of the call, the next invocation of *PR_NewEntry()* will result in an ID of either (`id`+1) if the high watermark user ID was just set, or `id`-1) if the highest group ID was just set.

The underlying *ubik*-based PDB is manipulated under a write transaction, and is write-locked for the duration of the operation.

Error Codes

`PRPERM` The caller's identity could not be established, or they are not authorized to perform the operation.

`PRDBBAD` An error was found relating to the PDB.

3.6.15 PR_ListElements — List all IDs describign user/group membership

```
int PR_ListElements(IN struct rx_connection *z_conn,
                    IN long id,
                    OUT prlist *elist,
                    OUT long *over)
```

Description

This interface function returns all of the IDs associated with the PDB entry whose ID is specified in the `id` argument. Basically, it returns the meaningful contents of the `entries` array in the associated `struct prentry`, along with the meaningful contents of the `entries` array from each chained continuation block (of type `struct contentry`). For a user `id`, the entries on this list represent the set of groups of which the user is a member. For a group `id`, the entries signify the list of users belonging to that group.

The IDs that appear in the `elist` list are sorted. If the size of the generated list has overflowed the maximum list size, `PR_MAXGROUPS` (5,000), then the `over` argument is set to a non-zero value.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

`PRPERM` The caller's identity could not be established, or is not authorized to perform the operation.

`PRNOENT` A PDB entry matching `id` was not found.

`PRDBBAD` An error was found relating to the PDB.

This interface function is closely related to the *PR_GetCPS()* routine. See Section 3.6.16 for a comparison of the two.

3.6.16 PR_GetCPS — Get the CPS for the given ID

```
int PR_GetCPS(IN struct rx_connection *z_conn,
              IN long id,
              OUT prlist *elist,
              OUT long *over)
```

Description

Generate the *Current Protection Subdomain*, or *CPS*, for the AFS user with the given numerical `id`. Basically, the CPS is the closure of all identities that the given `id` can use. This list of groups and special AFS identities (e.g. `ANYUSERID` and `AUTHUSERID`) to which the specified user belongs is placed in the `elist` structure passed in as an argument to this routine. The user's own ID also appears on this list.

The IDs that appear in the `elist` list are sorted. If the size of the list generated for the CPS has overflowed the maximum list size, `PR_MAXGROUPS` (5,000), then the `over` argument is set to a non-zero value.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation. As generating the CPS is a potentially lengthy operation, the lightweight *Protection Server* thread executing this call offers itself for rescheduling periodically. After processing the main PDB entry, the thread allows other active threads to run after handling every group of 50 IDs in a continuation block. This threading activity is completely invisible to the caller.

Error Codes

`PRNOENT` A PDB entry matching `id` was not found.

`PRPERM` The caller's identity could not be established, or is not authorized to perform the operation.

`PRDBBAD` An error was found relating to the PDB.

3.6.17 PR_ListOwned — List all IDs owned by the given ID

```
int PR_ListOwned(IN struct rx_connection *z_conn,
                 IN long id,
                 OUT prlist *elist,
                 OUT long *over)
```

Description

This function lists the set of AFS IDs owned by the user or group entity whose ID is `id`. Basically, all IDs appearing in the linked list hanging from the `owned` field in the entity's associated PDB entry are included. If the `id` parameter is set to zero, then the PDB orphan list is constructed. In this case, the IDs are taken from the list hanging from the `orphan` field from the PDB's header. In either case, the resulting list is placed in the variable-length `elist` array.

The IDs that appear in the `elist` list are sorted. If the size of the generated list has overflowed the maximum list size, `PR_MAXGROUPS` (5,000), then the `over` argument is set to a non-zero value.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

`PRPERM` The caller's identity could not be established, or is not authorized to perform the operation.

`PRNOENT` A PDB entry matching `id` was not found.

`PRDBBAD` An error was found relating to the PDB.

3.6.18 PR_IsAMemberOf — Is a given user ID a member of a specified group?

```
int PR_IsAMemberOf(IN struct rx_connection *z_conn,
                   IN long uid,
                   IN long gid,
                   OUT long *flag)
```

Description

This function sets its `flag` argument to a non-zero value if the user whose numerical ID is `uid` is a member of the `gid` group.

The underlying *ubik*-based PDB is manipulated under a read transaction, and is read-locked for the duration of the operation.

Error Codes

- `PRPERM` The caller's identity could not be established, or is not authorized to perform the operation.
- `PRNOENT` A PDB entry matching `uid` or `gid` was not found.
- `PRBADARG` The given `uid` is not an identifier for a user, or the given `gid` is not an identifier for a group.
- `PRDBBAD` An error was found relating to the PDB.

Bibliography

[1] Transarc Corporation, *AFS-3 Programmer's Reference: BOS Server Interface*, FS-00-D161, Pittsburgh, PA, June 1991

[2] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[3] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[4] Zayas, Edward R., Transarc Corporation. *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*, FS-00-D164, Pittsburgh, PA, April 1991.

[5] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[6] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Ubik - A Library For Managing Ubiquitous Data*, ITCID, Pittsburgh, PA, Month, 1988.

[7] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Quorum Completion*, ITCID, Pittsburgh, PA, Month, 1988.

Volume Server/VL Server Interface

Chapter 1

Overview

1.1 Introduction

This document describes the architecture and interfaces for two of the important agents of the AFS distributed file system, the *Volume Server* and the *Volume Location Server*. The *Volume Server* allows operations affecting entire AFS *volumes* to be executed, while the *Volume Location Server* provides a lookup service for volumes, identifying the server or set of servers on which volume instances reside.

1.2 Volumes

1.2.1 Definition

The underlying concept manipulated by the two AFS servers examined by this document is the *volume*. Volumes are the basic mechanism for organizing the data stored within the file system. They provide the foundation for addressing, storing, and accessing file data, along with serving as the administrative units for replication, backup, quotas, and data motion between *File Server*s.

Specifically, a volume is a container for a hierarchy of files, a connected file system subtree. In this respect, a volume is much like a traditional UNIX file system partition. Like a partition, a volume can be *mounted* in the sense that the *root directory* of the volume can be named within another volume at an AFS mount point. The entire file system hierarchy is built up in this manner, using mount points to glue together the individual subtrees resident within each volume. The root of this hierarchy is then mounted by each AFS client machine using a conventional UNIX mount point within the workstation's local file system. By convention, this entryway into the AFS domain is mounted on the */afs* local directory. From a user's point of view, there is only a single mount point to the system; the internal mount points are generally transparent.

1.2.2 Volume Naming

There are two methods by which volumes may be named. The first is via a human-readable string name, and the second is via a 32-bit numerical identifier. Volume identifiers, whether string or numerical, must be unique within any given cell. AFS mount points may use either representation to specify the volume whose root directory is to be accessed at the given position. Internally, however, AFS agents use the numerical form of identification exclusively, having to translate names to the corresponding 32-bit value.

1.2.3 Volume Types

There are three basic volume types: read-write, read-only, and backup volumes.

- **Read-write:** The data in this volume may be both read and written by those clients authorized to do so.
- **Read-only:** It is possible to create one or more read-only snapshots of read-write volumes. The read-write volume serving as the source image is referred to as the *parent volume*. Each read-only *clone*, or *child*, instance must reside on a different UNIX disk partition than the other clones. Every clone instance generated from the same parent read-write volume has the identical volume name and numerical volume ID. This is the reason why no two clones may appear on the same disk partition, as there would be no way to differentiate the two. AFS clients are allowed to read files and directories from read-only volumes, but cannot overwrite them individually. However, it *is* possible to make changes to the read-write parent and then *release* the contents of the entire volume to all the read-only replicas. The release operation fails if it does not reach the appropriate replication sites.
- **Backup:** A backup volume is a special instance of a read-only volume. While it is also a read-only snapshot of a given read-write volume, only one instance is allowed to exist at any one time. Also, the backup volume must reside on the same partition as the parent read-write volume from which it was created. It is from a backup volume that the AFS backup system writes file system data to tape. In

addition, backup volumes may be mounted into the file tree just like the other volume types. In fact, by convention, the backup volume for each user's home directory subtree is typically mounted as *OldFiles* in that directory. If a user accidentally deletes a file that resides in the backup snapshot, the user may simply copy it out of the backup directly without the assistance of a system administrator, or any kind of tape restore operation.

Backup volume are implemented in a copy-on-write fashion. Thus, backup volumes may be envisioned as consisting of a set of pointers to the true data objects in the base read-write volume when they are first created. When a file is overwritten in the read-write version for the first time after the backup volume was created, the original data is physically written to the backup volume, breaking the copy-on-write link. With this mechanism, backup volumes maintain the image of the read-write volume at the time the snapshot was taken using the minimum amount of additional disk space.

1.3 Scope

This paper is a member of a documentation suite providing specifications of the operation and interfaces offered by the various AFS servers and agents. The scope of this work is to provide readers with a sufficiently detailed description of the *Volume Location Server* and the *Volume Server* so that they may construct client applications which call their RPC interface routines.

1.4 Document Layout

After this introductory portion of the document, Chapters 2 and 3 examine the architecture and RPC interface of the *Volume Location Server* and its replicated database. Similarly, Chapters 4 and 5 describe the architecture and RPC interface of the *Volume Server*.

Chapter 2

Volume Location Server **Architecture**

2.1 Introduction

The *Volume Location Server* allows AFS agents to query the location and basic status of volumes resident within the given cell. *Volume Location Server* functions may be invoked directly from authorized users via the `vos` utility.

This chapter briefly discusses various aspects of the *Volume Location Server*'s architecture. First, the need for volume location is examined, and the specific parties that call the *Volume Location Server* interface routines are identified. Then, the database maintained to provide volume location service, the Volume Location Database (VLDB), is examined. Finally, the *vlserver* process which implements the *Volume Location Server* is considered.

As with all AFS servers, the *Volume Location Server* uses the *Rx* remote procedure call package for communication with its clients.

2.2 The Need For Volume Location

The *Cache Manager* agent is the primary consumer of AFS volume location service, on which it is critically dependent for its own operation. The *Cache Manager* needs to map volume names or numerical identifiers to the set of *File Server*s on which its instances reside in order to satisfy the file system requests it is processing on behalf of it clients. Each time a *Cache Manager* encounters a mount point for which it does not have location information cached, it must acquire this information before the pathname resolution may be successfully completed. Once the *File Server* set is known for a particular volume, the *Cache Manager* may then select the proper site among them (e.g. choosing the single home for a read-write volume, or randomly selecting a site from a read-only volume's replication set) and begin addressing its file manipulation operations to that specific server.

While the *Cache Manager* consults the volume location service, it is not capable of *changing* the location of volumes and hence modifying the information contained therein. This capability to perform acts which change volume location is concentrated within the *Volume Server*. The *Volume Server* process running on each server machine manages all volume operations affecting that platform, including creations, deletions, and movements between servers. It must update the volume location database every time it performs one of these actions.

None of the other AFS system agents has a need to access the volume location database for its site. Surprisingly, this also applies to the *File Server* process. It is only aware of the specific set of volumes that reside on the set of physical disks directly attached to the machine on which they execute. It has no knowlege of the universe of volumes resident on other servers, either within its own cell or in foreign cells.

2.3 The VLDB

The Volume Location Database (VLDB) is used to allow AFS application programs to discover the location of any volume within its cell, along with select information about the nature and state of that volume. It is organized in a very straightforward fashion, and uses the *ubik* [4] [5] facility to to provide replication across multiple server sites.

2.3.1 Layout

The VLDB itself is a very simple structure, and synchronized copies may be maintained at two or more sites. Basically, each copy consists of header information, followed by a linear (yet unbounded) array of entries. There are several associated hash tables used to perform lookups into the VLDB. The first hash table looks up volume location information based on the volume's name. There are three other hash tables used for lookup, based on volume ID/type pairs, one for each possible volume type.

The VLDB for a large site may grow to contain tens of thousands of entries, so some attempts were made to make each entry as small as possible. For example, server addresses within VLDB entries are represented as single-byte indicies into a table containing the full longword IP addresses.

A free list is kept for deleted VLDB entries. The VLDB will not grow unless all the entries on the free list have been exhausted, keeping it as compact as possible.

2.3.2 Database Replication

The VLDB, along with other important AFS databases, may be replicated to multiple sites to improve its availability. The *ubik* replication package is used to implement this functionality for the VLDB. A full description of *ubik* and of the *quorum completion* algorithm it implements may be found in [4] and [5]. The basic abstraction provided by *ubik* is that of a disk file replicated to multiple server locations. One machine is considered to be the *synchronization site*, handling all write operations on the database file. Read operations may be directed to any of the active members of the *quorum*, namely a subset of the replication sites large enough to insure integrity across such failures as individual server crashes and network partitions. All of the quorum members participate in regular *elections* to determine the current synchronization site. The *ubik* algorithms allow server machines to enter and exit the quorum in an orderly and consistent fashion. All operations to one of these replicated "abstract files" are performed as part of a *transaction*. If all the related operations performed under a transaction are successful, then the transaction is *committed*, and the changes are made permanent. Otherwise, the transaction is *aborted*, and all of the operations for that transaction are undone.

2.4 The *vlserver* Process

The user-space *vlserver* process is in charge of providing volume location service for AFS clients. This program maintains the VLDB replica at its particular server, and cooperates with all other *vlserver* processes running in the given cell to propagate updates to the database. It implements the RPC interface defined in the *vldbint.xg* definition file for the *rxgen* RPC stub generator program. As part of its startup sequence, it must discover the VLDB version it has on its local disk, move to join the quorum of replication sites for the VLDB, and get the latest version if the one it came up with was out of date. Eventually, it will synchronize with the other VLDB replication sites, and it will begin accepting calls.

The *vlserver* program uses at most three *Rx* worker threads to listen for incoming *Volume Location Server* calls. It has a single, optional command line argument. If the string "`-noauth`" appears when the program is invoked, then *vlserver* will run in an unauthenticated mode where any individual is considered authorized to perform any VLDB operation. This mode is necessary when first bootstrapping an AFS installation.

Chapter 3

Volume Location Server **Interface**

3.1 Introduction

This chapter documents the API for the *Volume Location Server* facility, as defined by the vldbint.xg *Rxgen* interface file and the vldbint.h include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer appear here.

It is expected that *Volume Location Server* client programs run in user space, as does the associated `vos` volume utility. However, the kernel-resident *Cache Manager* agent also needs to call a subset of the *Volume Location Server*'s RPC interface routines. Thus, a second *Volume Location Server* interface is available, built exclusively to satisfy the *Cache Manager*'s limited needs. This subset interface is defined by the *afsvlint.xg Rxgen* interface file, and is examined in the final section of this chapter.

3.2 Constants

This section covers the basic constant definitions of interest to the *Volume Location Server* application programmer. These definitions appear in the *vldbint.h* file, automatically generated from the *vldbint.xg Rxgen* interface file, and in *vlserver.h*.

Each subsection is devoted to describing the constants falling into the following categories:

- Configuration and boundary quantities
- Update entry bits
- List-by-attribute bits
- Volume type indices
- States for `struct vlentry`
- States for `struct vldbentry`
- `ReleaseType` argument values
- Miscellaneous items

3.2.1 Configuration and Boundary Quantities

These constants define some basic system values, including configuration information.

Name	*Value*	*Description*
`MAXNAMELEN`	65	Maximum size of various character strings, including volume name fields in structures and host names.
`MAXNSERVERS`	8	Maximum number of replication sites for a volume.
`MAXTYPES`	3	Maximum number of volume types.
`VLDBVERSION`	1	VLDB database version number.
`HASHSIZE`	8,191	Size of internal *Volume Location Server* volume name and volume ID hash tables. This must always be a prime number.
`NULLO`	0	Specifies a null pointer value.
`VLDBALLOCCOUNT`	40	Value used when allocating memory internally for VLDB entry records.
`BADSERVERID`	255	Illegal *Volume Location Server* host ID.
`MAXSERVERID`	30	Maximum number of servers appearing in the VLDB.
`MAXSERVERFLAG`	0x80	First unused flag value in such fields as `serverFlags` in `struct vldbentry` and `RepsitesNewFlags` in `struct VldbUpdateEntry` .
`MAXPARTITIONID`	126	Maximum number of AFS disk partitions for any one server.
`MAXBUMPCOUNT`	0x7fffffff	Maximum interval that the current high-watermark value for a volume ID can be increased in one operation.
`MAXLOCKTIME`	0x7fffffff	Maximum number of seconds that any VLDB entry can remained locked.
`SIZE`	1,024	Maximum size of the `name` field within a `struct VolumeDiskData` inside a `struct Volume`'s `header` field.

3.2.2 Update Entry Bits

These constants define bit values for the `Mask` field in the `struct VldbUpdateEntry`. Specifically, setting these bits is equivalent to declaring that the corresponding field within an object of type `struct VldbUpdateEntry` has been set. For example, setting the `VLUPDATE_VOLUMENAME` flag in `Mask` indicates that the `name` field contains a valid value.

Name	*Value*	*Description*
VLUPDATE_VOLUMENAME	0x0001	If set, indicates that the name field is valid.
VLUPDATE_VOLUMETYPE	0x0002	If set, indicates that the volumeType field is valid.
VLUPDATE_FLAGS	0x0004	If set, indicates that the flags field is valid.
VLUPDATE_READONLYID	0x0008	If set, indicates that the ReadOnlyId field is valid.
VLUPDATE_BACKUPID	0x0010	If set, indicates that the BackupId field is valid.
VLUPDATE_REPSITES	0x0020	If set, indicates that the nModifiedRepsites field is valid.
VLUPDATE_CLONEID	0x0080	If set, indicates that the cloneId field is valid.
VLUPDATE_REPS_DELETE	0x0100	Is the replica being deleted?
VLUPDATE_REPS_ADD	0x0200	Is the replica being added?
VLUPDATE_REPS_MODSERV	0x0400	Is the server part of the replica location correct?
VLUPDATE_REPS_MODPART	0x0800	Is the partition part of the replica location correct?
VLUPDATE_REPS_MODFLAG	0x1000	Various modification flag values.

3.2.3 List-By-Attribute Bits

These constants define bit values for the `Mask` field in the `struct VldbListByAttributes` is to be used in a match. Specifically, setting these bits is equivalent to declaring that the corresponding field within an object of type `struct VldbListByAttributes` is set. For example, setting the `VLLIST_SERVER` flag in `Mask` indicates that the `server` field contains a valid value.

Name	*Value*	*Description*
VLLIST_SERVER	0x1	If set, indicates that the server field is valid.
VLLIST_PARTITION	0x2	If set, indicates that the partition field is valid.
VLLIST_VOLUMETYPE	0x4	If set, indicates that the volumetype field is valid.
VLLIST_VOLUMEID	0x8	If set, indicates that the volumeid field is valid.
VLLIST_FLAG	0x10	If set, indicates that the flag field is valid.

3.2.4 Volume Type Indices

These constants specify the order of entries in the `volumeid` array in an object of type `struct vldbentry`. They also identify the three different types of volumes in AFS.

Name	*Value*	*Description*
RWVOL	0	Read-write volume.
ROVOL	1	Read-only volume.
BACKVOL	2	Backup volume.

3.2.5 States for `struct vlentry`

The following constants appear in the `flags` field in objects of type `struct vlentry`. The first three values listed specify the state of the entry, while all the rest stamp the entry with the type of an ongoing volume operation, such as a move, clone, backup, deletion, and dump. These volume operations are the legal values to provide to the `voloper` parameter of the *VL_SetLock()* interface routine.

Name	*Value*	*Description*
VLFREE	0x1	Entry is in the free list.
VLDELETED	0x2	Entry is soft-deleted.
VLLOCKED	0x4	Advisory lock held on the entry.
VLOP_MOVE	0x10	The associated volume is being moved between servers.
VLOP_RELEASE	0x20	The associated volume is being cloned to its replication sites.
VLOP_BACKUP	0x40	A backup volume is being created for the associated volume.
VLOP_DELETE	0x80	The associated volume is being deleted.
VLOP_DUMP	0x100	A dump is being taken of the associated volume.

For convenience, the constant `VLOP_ALLOPERS` is defined as the inclusive `OR` of the above values from `VLOP_MOVE` through `VLOP_DUMP`.

3.2.6 States for `struct vldbentry`

Of the following constants, the first three appear in the `flags` field within an object of type `struct vldbentry`, advising of the existence of the basic volume types for the given volume, and hence the validity of the entries in the `volumeId` array field. The rest of the values provided in this table appear in the `serverFlags` array field, and apply to the instances of the volume appearing in the various replication sites.

This structure appears in numerous *Volume Location Server* interface calls, namely *VL_CreateEntry()*, *VL_GetEntryByID()*, *VL_GetEntryByName()*, *VL_ReplaceEntry()* and *VL_ListEntry()*.

Name	*Value*	*Description*
`VLF_RWEXISTS`	0x1000	The read-write volume ID is valid.
`VLF_ROEXISTS`	0x2000	The read-only volume ID is valid.
`VLF_BACKEXISTS`	0x4000	The backup volume ID is valid.
`VLSF_NEWREPSITE`	0x01	Not used; originally intended to mark an entry as belonging to a partially-created volume instance.
`VLSF_ROVOL`	0x02	A read-only version of the volume appears at this server.
`VLSF_RWVOL`	0x04	A read-write version of the volume appears at this server.
`VLSF_BACKVOL`	0x08	A backup version of the volume appears at this server.

3.2.7 `ReleaseType` Argument Values

The following values are used in the `ReleaseType` argument to various *Volume Location Server* interface routines, namely *VL_ReplaceEntry()*, *VL_UpdateEntry()* and *VL_ReleaseLock()*.

Name	*Value*	*Description*
`LOCKREL_TIMESTAMP`	1	Is the `LockTimestamp` field valid?
`LOCKREL_OPCODE`	2	Are any of the bits valid in the `flags` field?
`LOCKREL_AFSID`	4	Is the `LockAfsId` field valid?

3.2.8 Miscellaneous

Miscellaneous values.

Name	*Value*	*Description*
`VLREPSITE_NEW`	1	Has a replication site gotten a new release of a volume?

A synonym for this constant is `VLSF_NEWREPSITE`.

3.3 Structures and Typedefs

This section describes the major exported *Volume Location Server* data structures of interest to application programmers, along with the typedefs based upon those structures.

3.3.1 `struct vldbentry`

This structure represents an entry in the VLDB as made visible to *Volume Location Server* clients. It appears in numerous *Volume Location Server* interface calls, namely *VL_CreateEntry()*, *VL_GetEntryByID()*, *VL_GetEntryByName()*, *VL_ReplaceEntry()* and *VL_ListEntry()*.

Fields

char name[] - The string name for the volume, with a maximum length of `MAXNAMELEN` (65) characters, including the trailing null.

long volumeType - The volume type, one of `RWVOL`, `ROVOL`, or `BACKVOL`.

long nServers - The number of servers that have an instance of this volume.

long serverNumber[] - An array of indices into the table of servers, identifying the sites holding an instance of this volume. There are at most `MAXNSERVERS` (8) of these server sites allowed by the *Volume Location Server*.

long serverPartition[] - An array of partition identifiers, corresponding directly to the `serverNumber` array, specifying the partition on which each of those volume instances is located. As with the `serverNumber` array, `serverPartition` has up to `MAXNSERVERS` (8) entries.

long serverFlags[] - This array holds one flag value for each of the servers in the previous arrays. Again, there are `MAXNSERVERS` (8) slots in this array.

u_long volumeId[] - An array of volume IDs, one for each volume type. There are `MAXTYPES` slots in this array.

long cloneId - This field is used during a cloning operation.

long flags - Flags concerning the status of the fields within this structure; see Section 3.2.6 for the bit values that apply.

3.3.2 struct vlentry

This structure is used internally by the *Volume Location Server* to fully represent a VLDB entry. The client-visible `struct vldbentry` represents merely a subset of the information contained herein.

Fields

u_long volumeId[] - An array of volume IDs, one for each of the `MAXTYPES` of volume types.

long flags - Flags concerning the status of the fields within this structure; see Section 3.2.6 for the bit values that apply.

long LockAfsId - The individual who locked the entry. This feature has not yet been implemented.

long LockTimestamp - Time stamp on the entry lock.

long cloneId - This field is used during a cloning operation.

long AssociatedChain - Pointer to the linked list of associated VLDB entries.

long nextIdHash[] - Array of `MAXTYPES` next pointers for the ID hash table pointer, one for each related volume ID.

long nextNameHash - Next pointer for the volume name hash table.

long spares1[] - Two longword spare fields.

char name[] - The volume's string name, with a maximum of `MAXNAMELEN` (65) characters, including the trailing null.

u_char volumeType - The volume's type, one of `RWVOL`, `ROVOL`, or `BACKVOL`.

u_char serverNumber[] - An array of indices into the table of servers, identifying the sites holding an instance of this volume. There are at most `MAXNSERVERS` (8) of these server sites allowed by the *Volume Location Server*.

u_char serverPartition[] - An array of partition identifiers, corresponding directly to the `serverNumber` array, specifying the partition on which each of those volume instances is located. As with the `serverNumber` array, `serverPartition` has up to `MAXNSERVERS` (8) entries.

u_char serverFlags[] - This array holds one flag value for each of the servers in the previous arrays. Again, there are `MAXNSERVERS` (8) slots in this array.

u_char RefCount - Only valid for read-write volumes, this field serves as a reference count, basically the number of dependent children volumes.

char spares2[] - This field is used for 32-bit alignment.

3.3.3 struct vital_vlheader

This structure defines the leading section of the VLDB header, of type `struct vlheader`. It contains frequently-used global variables and general statistics information.

Fields

long vldbversion - The VLDB version number. This field must appear first in the structure.

long headersize - The total number of bytes in the header.

long freePtr - Pointer to the first free enry in the free list, if any.

long eofPtr - Pointer to the first free byte in the header file.

long allocs - The total number of calls to the internal *AllocBlock()* function directed at this file.

long frees - The total number of calls to the internal *FreeBlock()* function directed at this file.

long MaxVolumeId - The largest volume ID ever granted for this cell.

long totalEntries[] - The total number of VLDB entries by volume type in the VLDB. This array has `MAXTYPES` slots, one for each volume type.

3.3.4 struct vlheader

This is the layout of the information stored in the VLDB header. Notice it includes an object of type `struct vital_vlheader` described above (see Section 3.3.3) as the first field.

Fields

struct vital_vlheader vital_header - Holds critical VLDB header information.

u_long IpMappedAddr[] - Keeps `MAXSERVERID+1` mappings of IP addresses to relative ones.

long VolnameHash[] - The volume name hash table, with `HASHSIZE` slots.

long VolidHash[][] - The volume ID hash table. The first dimension in this array selects which of the `MAXTYPES` volume types is desired, and the second dimension actually implements the `HASHSIZE` hash table buckets for the given volume type.

3.3.5 struct VldbUpdateEntry

This structure is used as an argument to the *VL_UpdateEntry()* routine (see Section 3.6.7). Please note that multiple entries can be updated at once by setting the appropriate `Mask` bits. The bit values for this purpose are defined in Section 3.2.2.

Fields

u_long Mask - Bit values determining which fields are to be affected by the update operation.

char name[] - The volume name, up to `MAXNAMELEN` (65) characters including the trailing null.

long volumeType - The volume type.

long flags - This field is used in conjuction with `Mask` (in fact, one of the `Mask` bits determines if this field is valid) to choose the valid fields in this record.

u_long ReadOnlyId - The read-only ID.

u_long BackupId - The backup ID.

long cloneId - The clone ID.

long nModifiedRepsites - Number of replication sites whose entry is to be changed as below.

u_long RepsitesMask[] - Array of bit masks applying to the up to `MAXNSERVERS` (8) replication sites involved.

long RepsitesTargetServer[] - Array of target servers for the operation, at most `MAXNSERVERS` (8) of them.

long RepsitesTargetPart[] - Array of target server partitions for the operation, at most `MAXNSERVERS` (8) of them.

long RepsitesNewServer[] - Array of new server sites, at most `MAXNSERVERS` (8) of them.

long RepsitesNewPart[] - Array of new server partitions for the operation, at most `MAXNSERVERS` (8) of them.

long RepsitesNewFlags[] - Flags applying to each of the new sites, at most `MAXNSERVERS` (8) of them.

3.3.6 struct VldbListByAttributes

This structure is used by the *VL_ListAttributes()* routine (see Section 3.6.11).

Fields

u_long Mask - Bit mask used to select the following attribute fields on which to match.

long server - The server address to match.

long partition - The partition ID to match.

long volumetype - The volume type to match.

long volumeid - The volume ID to match.

long flag - Flags concerning these values.

3.3.7 `struct single_vldbentry`

This structure is used to construct the `vldblist` object (See Section 3.3.12), which basically generates a queueable (singly-linked) version of `struct vldbentry`.

Fields

vldbentry VldbEntry - The VLDB entry to be queued.

vldblist next_vldb - The next pointer in the list.

3.3.8 `struct vldb_list`

This structure defines the item returned in linked list form from the *VL_LinkedList()* function (see Section 3.6.12). This same object is also returned in bulk form in calls to the *VL_ListAttributes()* routine (see Section 3.6.11).

Fields

vldblist node - The body of the first object in the linked list.

3.3.9 `struct vldstats`

This structure defines fields to record statistics on opcode hit frequency. The `MAX_NUMBER_OPCODES` constant has been defined as the maximum number of opcodes supported by this structure, and is set to 30.

Fields

unsigned long start_time - Clock time when opcode statistics were last cleared.

long requests[] - Number of requests received for each of the `MAX_NUMBER_OPCODES` opcode types.

long aborts[] - Number of aborts experienced for each of the `MAX_NUMBER_OPCODES` opcode types.

long reserved[] - These five longword fields are reserved for future use.

3.3.10 bulk

```
typedef opaque bulk<DEFAULTBULK>;
```

This typedef may be used to transfer an uninterpreted set of bytes across the *Volume Location Server* interface. It may carry up to `DEFAULTBULK` (10,000) bytes.

Fields

`bulk_len` - The number of bytes contained within the data pointed to by the next field.

`bulk_val` - A pointer to a sequence of `bulk_len` bytes.

3.3.11 bulkentries

```
typedef vldbentry bulkentries<>;
```

This typedef is used to transfer an unbounded number of `struct vldbentry` objects. It appears in the parameter list for the *VL_ListAttributes()* interface function.

Fields

`bulkentries_len` - The number of `vldbentry` structures contained within the data pointed to by the next field.

`bulkentries_val` - A pointer to a sequence of `bulkentries_len vldbentry` structures.

3.3.12 vldblist

```
typedef struct single_vldbentry *vldblist;
```

This typedef defines a queueable `struct vldbentry` object, referenced by the `single_vldbentry` typedef as well as `struct vldb_list`.

3.3.13 vlheader

```
typedef struct vlheader vlheader;
```

This typedef provides a short name for objects of type `struct vlheader` (see Section 3.3.4).

3.3.14 vlentry

```
typedef struct vlentry vlentry;
```

This typedef provides a short name for objects of type `struct vlentry` (see Section 3.3.2).

3.4 Error Codes

This section covers the set of error codes exported by the *Volume Location Server*, displaying the printable phrases with which they are associated.

Name	*Value*	*Description*
VL_IDEXIST	(363520L)	Volume Id entry exists in vl database.
VL_IO	(363521L)	I/O related error.
VL_NAMEEXIST	(363522L)	Volume name entry exists in vl database.
VL_CREATEFAIL	(363523L)	Internal creation failure.
VL_NOENT	(363524L)	No such entry.
VL_EMPTY	(363525L)	Vl database is empty.
VL_ENTDELETED	(363526L)	Entry is deleted (soft delete).
VL_BADNAME	(363527L)	Volume name is illegal.
VL_BADINDEX	(363528L)	Index is out of range.
VL_BADVOLTYPE	(363529L)	Bad volume type.
VL_BADSERVER	(363530L)	Illegal server number (out of range).
VL_BADPARTITION	(363531L)	Bad partition number.
VL_REPSFULL	(363532L)	Run out of space for Replication sites.
VL_NOREPSERVER	(363533L)	No such Replication server site exists.
VL_DUPREPSERVER	(363534L)	Replication site already exists.
VL_RWNOTFOUND	(363535L)	Parent R/W entry not found.
VL_BADREFCOUNT	(363536L)	Illegal Reference Count number.
VL_SIZEEXCEEDED	(363537L)	Vl size for attributes exceeded.
VL_BADENTRY	(363538L)	Bad incoming vl entry.
VL_BADVOLIDBUMP	(363539L)	Illegal max volid increment.
VL_IDALREADYHASHED	(363540L)	RO/BACK id already hashed.
VL_ENTRYLOCKED	(363541L)	Vl entry is already locked.
VL_BADVOLOPER	(363542L)	Bad volume operation code.
VL_BADRELLOCKTYPE	(363543L)	Bad release lock type.
VL_RERELEASE	(363544L)	Status report: last release was aborted.
VL_BADSERVERFLAG	(363545L)	Invalid replication site server flag.
VL_PERM	(363546L)	No permission access.
VL_NOMEM	(363547L)	malloc(realloc) failed to alloc enough memory.

3.5 Macros

The *Volume Location Server* defines a small number of macros, as described in this section. They are used to update the internal statistics variables and to compute offsets into character strings. All of these macros really refer to internal operations, and strictly speaking should not be exposed in this interface.

3.5.1 *COUNT_REQ()*

```
#define COUNT_REQ(op)
    static int this_op = op-VL_LOWEST_OPCODE;
    dynamic_statistics.requests[this_op]++
```

Bump the appropriate entry in the variable maintaining opcode usage statistics for the *Volume Location Server*. Note that a static variable is set up to record `this_op`, namely the index into the opcode monitoring array. This static variable is used by the related *COUNT_ABO()* macro defined below.

3.5.2 *COUNT_ABO()*

```
#define COUNT_ABO dynamic_statistics.aborts[this_op]++
```

Bump the appropriate entry in the variable maintaining opcode abort statistics for the *Volume Location Server*. Note that this macro does not take any arguemnts. It expects to find a `this_op` variable in its environment, and thus depends on its related macro, *COUNT_REQ()* to define that variable.

3.5.3 *DOFFSET()*

```
#define DOFFSET(abase, astr, aitem)
    ((abase)+(((char *)(aitem)) - ((char *)(astr))))
```

Compute the byte offset of charcter object `aitem` within the enclosing object `astr`, also expressed as a character-based object, then offset the resulting address by `abase`. This macro is used ot compute locations within the VLDB when actually writing out information.

3.6 Functions

This section covers the *Volume Location Server* RPC interface routines. The majority of them are generated from the *vldbint.xg Rxgen* file, and are meant to be used by user-space agents. There is also a subset interface definition provided in the *afsvlint.xg Rxgen* file. These routines, described in Section 3.7, are meant to be used by a kernel-space agent when dealing with the *Volume Location Server*; in particular, they are called by the *Cache Manager*.

3.6.1 VL_CreateEntry — Create a VLDB entry

```
int VL_CreateEntry(IN struct rx_connection *z_conn,
                   IN vldbentry *newentry)
```

Description

This function creates a new entry in the VLDB, as specified in the `newentry` argument. Both the name and numerical ID of the new volume must be unique (e.g., it must not already appear in the VLDB). For non-read-write entries, the read-write parent volume is accessed so that its reference count can be updated, and the new entry is added to the parent's chain of associated entries.

The VLDB is write-locked for the duration of this operation.

Error Codes

- `VL_PERM` The caller is not authorized to execute this function.
- `VL_NAMEEXIST` The volume name already appears in the VLDB.
- `VL_CREATEFAIL` Space for the new entry cannot be allocated within the VLDB.
- `VL_BADNAME` The volume name is invalid.
- `VL_BADVOLTYPE` The volume type is invalid.
- `VL_BADSERVER` The indicated server information is invalid.
- `VL_BADPARTITION` The indicated partition information is invalid.
- `VL_BADSERVERFLAG` The server flag field is invalid.
- `VL_IO` An error occurred while writing to the VLDB.

3.6.2 VL_DeleteEntry — Delete a VLDB entry

```
int VL_DeleteEntry(IN struct rx_connection *z_conn,
                   IN long Volid,
                   IN long voltype)
```

Description

Delete the entry matching the given volume identifier and volume type as specified in the `Volid` and `voltype` arguments. For a read-write entry whose reference count is greater than 1, the entry is not actually deleted, since at least one child (read-only or backup) volume still depends on it. For cases of non-read-write volumes, the parent's reference count and associated chains are updated.

If the associated VLDB entry is already marked as deleted (i.e., its `flags` field has the `VLDELETED` bit set), then no further action is taken, and `VL_ENTDELETED` is returned. The VLDB is write-locked for the duration of this operation.

Error Codes

- `VL_PERM` The caller is not authorized to execute this function.
- `VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.
- `VL_NOENT` This volume instance does not appear in the VLDB.
- `VL_ENTDELETED` The given VLDB entry has already been marked as deleted.
- `VL_IO` An error occurred while writing to the VLDB.

3.6.3 VL_GetEntryByID — Get VLDB entry by volume ID/type

```
int VL_GetEntryByID(IN struct rx_connection *z_conn,
                    IN long Volid,
                    IN long voltype,
                    OUT vldbentry *entry)
```

Description

Given a volume's numerical identifier (`Volid`) and type (`voltype`), return a pointer to the entry in the VLDB describing the given volume instance.

The VLDB is read-locked for the duration of this operation.

Error Codes

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_NOENT` This volume instance does not appear in the VLDB.

`VL_ENTDELETED` The given VLDB entry has already been marked as deleted.

3.6.4 VL_GetEntryByName — Get VLDB entry by volume name

```
int VL_GetEntryByName(IN struct rx_connection *z_conn,
                      IN char *volumename,
                      OUT vldbentry *entry)
```

Description

Given the volume name in the `volumename` parameter, return a pointer to the entry in the VLDB describing the given volume. The name in `volumename` may be no longer than `MAXNAMELEN` (65) characters, including the trailing null. Note that it is legal to use the volume's numerical identifier (in string form) as the volume name.

The VLDB is read-locked for the duration of this operation.

This function is closely related to the *VL_GetEntryByID()* routine, as might be expected. In fact, the by-ID routine is called if the volume name provided in `volumename` is the string version of the volume's numerical identifier.

Error Codes

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_NOENT` This volume instance does not appear in the VLDB.

`VL_ENTDELETED` The given VLDB entry has already been marked as deleted.

`VL_BADNAME` The volume name is invalid.

3.6.5 VL_GetNewVolumeId — Generate a new volume ID

```
int VL_GetNewVolumeId(IN struct rx_connection *z_conn,
                      IN long bumpcount,
                      OUT long *newvolumid)
```

Description

Acquire `bumpcount` unused, consecutively-numbered volume identifiers from the *Volume Location Server*. The lowest-numbered of the newly-acquired set is placed in the `newvolumid` argument. The largest number of volume IDs that may be generated with any one call is bounded by the `MAXBUMPCOUNT` constant defined in Section 3.2.1. Currently, there is (effectively) no restriction on the number of volume identifiers that may thus be reserved in a single call.

The VLDB is write-locked for the duration of this operation.

Error Codes

`VL_PERM` The caller is not authorized to execute this function.

`VL_BADVOLIDBUMP` The value of the `bumpcount` parameter exceeds the system limit of `MAXBUMPCOUNT`.

`VL_IO` An error occurred while writing to the VLDB.

3.6.6 VL_ReplaceEntry — Replace entire contents of VLDB entry

```
int VL_ReplaceEntry(IN struct rx_connection *z_conn,
                    IN long Volid,
                    IN long voltype,
                    IN vldbentry *newentry,
                    IN long ReleaseType)
```

Description

Perform a wholesale replacement of the VLDB entry corresponding to the volume instance whose identifier is `Volid` and type `voltype` with the information contained in the `newentry` argument. Individual VLDB entry fields cannot be selectively changed while the others are preserved; *VL_UpdateEntry()* should be used for this objective. The permissible values for the `ReleaseType` parameter are defined in Section 3.2.7.

The VLDB is write-locked for the duration of this operation. All of the hash tables impacted are brought up to date to incorporate the new information.

Error Codes

`VL_PERM` The caller is not authorized to execute this function.

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_BADRELLOCKTYPE` An illegal release lock has been specified by the `ReleaseType` argument.

`VL_NOENT` This volume instance does not appear in the VLDB.

`VL_BADENTRY` An attempt was made to change a read-write volume ID.

`VL_IO` An error occurred while writing to the VLDB.

3.6.7 VL_UpdateEntry — Update contents of VLDB entry

```
int VL_UpdateEntry(IN struct rx_connection *z_conn,
                   IN long Volid,
                   IN long voltype,
                   IN VldbUpdateEntry *UpdateEntry,
                   IN long ReleaseType)
```

Description

Update the VLDB entry corresponding to the volume instance whose identifier is `Volid` and type `voltype` with the information contained in the `UpdateEntry` argument. Most of the entry's fields can be modified in a single call to *VL_UpdateEntry()*. The `Mask` field within the `UpdateEntry` parameter selects the fields to update with the values stored within the other `UpdateEntry` fields. Permissible values for the `ReleaseType` parameter are defined in Section 3.2.7.

The VLDB is write-locked for the duration of this operation.

Error Codes

`VL_PERM` The caller is not authorized to execute this function.

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_BADRELLOCKTYPE` An illegal release lock has been specified by the `ReleaseType` argument.

`VL_NOENT` This volume instance does not appear in the VLDB.

`VL_IO` An error occurred while writing to the VLDB.

3.6.8 VL_SetLock — Lock VLDB entry

```
int VL_SetLock(IN struct rx_connection *z_conn,
               IN long Volid,
               IN long voltype,
               IN long voloper)
```

Description

Lock the VLDB entry matching the given volume ID (`Volid`) and type (`voltype`) for volume operation `voloper` (e.g., `VLOP_MOVE` and `VLOP_RELEASE`). If the entry is currently unlocked, then its `LockTimestamp` will be zero. If the lock is obtained, the given `voloper` is stamped into the `flags` field, and the `LockTimestamp` is set to the time of the call.

Note: when the caller attempts to lock the entry for a release operation, special care is taken to abort the operation if the entry has already been locked for this operation, and the existing lock has timed out. In this case, `VL_SetLock()` returns `VL_RERELEASE`.

The VLDB is write-locked for the duration of this operation.

Error Codes

`VL_PERM` The caller is not authorized to execute this function.

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_BADVOLOPER` An illegal volume operation was specified in the `voloper` argument. Legal values are defined in the latter part of the table in Section 3.2.5.

`VL_ENTDELETED` The given VLDB entry has already been marked as deleted.

`VL_ENTRYLOCKED` The given VLDB entry has already been locked (which has not yet timed out).

`VL_RERELEASE` A VLDB entry locked for release has timed out, and the caller also wanted to perform a release operation on it.

`VL_IO` An error was experienced while attempting to write to the VLDB.

3.6.9 VL_ReleaseLock — Unlock VLDB entry

```
int VL_ReleaseLock(IN struct rx_connection *z_conn,
                   IN long Volid,
                   IN long voltype,
                   IN long ReleaseType)
```

Description

Unlock the VLDB entry matching the given volume ID (`Volid`) and type (`voltype`). The `ReleaseType` argument determines which VLDB entry fields from `flags` and `LockAfsId` will be cleared along with the lock timestamp in `LockTimestamp`. Permissible values for the `ReleaseType` parameter are defined in Section 3.2.7.

The VLDB is write-locked for the duration of this operation.

Error Codes

- `VL_PERM` The caller is not authorized to execute this function.
- `VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.
- `VL_BADRELLOCKTYPE` An illegal release lock has been specified by the `ReleaseType` argument.
- `VL_NOENT` This volume instance does not appear in the VLDB.
- `VL_ENTDELETED` The given VLDB entry has already been marked as deleted.
- `VL_IO` An error was experienced while attempting to write to the VLDB.

3.6.10 VL_ListEntry — Get contents of VLDB via index

```
int VL_ListEntry(IN struct rx_connection *z_conn,
                 IN long previous_index,
                 OUT long *count,
                 OUT long *next_index,
                 OUT vldbentry *entry)
```

Description

This function assists in the task of enumerating the contents of the VLDB. Given an index into the database, `previous_index`, this call return the single VLDB entry at that offset, placing it in the `entry` argument. The number of VLDB entries left to list is placed in `count`, and the index of the next entry to request is returned in `next_index`. If an illegal index is provided, `count` is set to -1.

The VLDB is read-locked for the duration of this operation.

Error Codes

`---` None.

3.6.11 VL_ListAttributes — List all VLDB entry matching given attributes, single return object

```
int VL_ListAttributes(IN struct rx_connection *z_conn,
                      IN VldbListByAttributes *attributes,
                      OUT long *nentries,
                      OUT bulkentries *blkentries)
```

Description

Retrieve all the VLDB entries that match the attributes listed in the `attributes` parameter, placing them in the `blkentries` object. The number of matching entries is placed in `nentries`. Matching can be done by server number, partition, volume type, flag, or volume ID. The legal values to use in the `attributes` argument are listed in Section 3.2.3. Note that if the `VLLIST_VOLUMEID` bit is set in `attributes`, all other bit values are ignored and the volume ID provided is the sole search criterion.

The VLDB is read-locked for the duration of this operation.

Note that *VL_ListAttributes()* is a potentially expensive function, as sequential search through all of the VLDB entries is performed in most cases.

Error Codes

`VL_NOMEM` Memory for the `blkentries` object could not be allocated.

`VL_NOENT` This specified volume instance does not appear in the VLDB.

`VL_SIZEEXCEEDED` Ran out of room in the `blkentries` object.

`VL_IO` Error while reading from the VLDB.

3.6.12 VL_LinkedList — List all VLDB entry matching given attributes, linked list return object

```
int VL_LinkedList(IN struct rx_connection *z_conn,
                  IN VldbListByAttributes *attributes,
                  OUT long *nentries,
                  OUT vldb_list *linkedentries)
```

Description

Retrieve all the VLDB entries that match the attributes listed in the `attributes` parameter, creating a linked list of entries based in the `linkedentries` object. The number of matching entries is placed in `nentries`. Matching can be done by server number, partition, volume type, flag, or volume ID. The legal values to use in the `attributes` argument are listed in Section 3.2.3. Note that if the `VLLIST_VOLUMEID` bit is set in `attributes`, all other bit values are ignored and the volume ID provided is the sole search criterion.

The *VL_LinkedList()* function is identical to the *VL_ListAttributes()*, except for the method of delivering the VLDB entries to the caller.

The VLDB is read-locked for the duration of this operation.

Error Codes

`VL_NOMEM` Memory for an entry in the list based at `linkedentries` object could not be allocated.

`VL_NOENT` This specified volume instance does not appear in the VLDB.

`VL_SIZEEXCEEDED` Ran out of room in the current list object.

`VL_IO` Error while reading from the VLDB.

3.6.13 VL_GetStats — Get *Volume Location Server* statistics

```
int VL_GetStats(IN struct rx_connection *z_conn,
                OUT vldstats *stats,
                OUT vital_vlheader *vital_header)
```

Description

Collect the different types of VLDB statistics. Part of the VLDB header is returned in `vital_header`, which includes such information as the number of allocations and frees performed, and the next volume ID to be allocated. The dynamic per-operation stats are returned in the `stats` argument, reporting the number and types of operations and aborts.

The VLDB is read-locked for the duration of this operation.

Error Codes

`VL_PERM` The caller is not authorized to execute this function.

3.6.14 VL_Probe — Verify *Volume Location Server* connectivity/status

```
int VL_Probe(IN struct rx_connection *z_conn)
```

Description

This routine serves a "pinging" function to determine whether the *Volume Location Server* is still running. If this call succeeds, then the *Volume Location Server* is shown to be capable of responding to RPCs, thus confirming connectivity and basic operation.

The VLDB is *not* locked for this operation.

Error Codes

--- None.

3.7 Kernel Interface Subset

The interface described by this document so far applies to user-level clients, such as the `vos` utility. However, some volume location operations must be performed from within the kernel. Specifically, the *Cache Manager* must find out where volumes reside and otherwise gather information about them in order to conduct its business with the *File Servers* holding them. In order to support *Volume Location Server* interconnection for agents operating within the kernel, the *afsvlint.xg Rxgen* interface was built. It is a minimal subset of the user-level *vldbint.xg* definition. Within *afsvlint.xg*, there are duplicate definitions for such constants as `MAXNAMELEN`, `MAXNSERVERS`, `MAXTYPES`, `VLF_RWEXISTS`, `VLF_ROEXISTS`, `VLF_BACKEXISTS`, `VLSF_NEWREPSITE`, `VLSF_ROVOL`, `VLSF_RWVOL`, and `VLSF_BACKVOL`. Since the only operations the *Cache Manager* must perform are volume location given a specific volume ID or name, and to find out about unresponsive *Volume Location Servers*, the following interface routines are duplicated in *afsvlint.xg*, along with the `struct vldbentry` declaration:

- *VL_GetEntryByID()*
- *VL_GetEntryByName()*
- *VL_Probe()*

Chapter 4

Volume Server **Architecture**

4.1 Introduction

The *Volume Server* allows administrative tasks and probes to be performed on the set of AFS volumes residing on the machine on which it is running. As described in Chapter 2, a distributed database holding volume location info, the VLDB, is used by client applications to locate these volumes. *Volume Server* functions are typically invoked either directly from authorized users via the `vos` utility or by the AFS backup system.

This chapter briefly discusses various aspects of the *Volume Server*'s architecture. First, the high-level on-disk representation of volumes is covered. Then, the transactions used in conjuction with volume operations are examined. Then, the program implementing the *Volume Server*, *volserver*, is considered. The nature and format of the log file kept by the *Volume Server* rounds out the description.

As with all AFS servers, the *Volume Server* uses the *Rx* remote procedure call package for communication with its clients.

4.2 Disk Representation

For each volume on an AFS partition, there exists a file visible in the UNIX name space which describes the contents of that volume. By convention, each of these files is named by concatenating a prefix string, "*V*", the numerical volume ID, and the postfix string "*.vol*". Thus, file *V0536870918.vol* describes the volume whose numerical ID is 0536870918. Internally, each per-volume descriptor file has such fields as a version number, the numerical volume ID, and the numerical parent ID (useful for read-only or backup volumes). It also has a list of related inodes, namely files which are *not* visible from the UNIX name space (i.e., they do not appear as entries in any UNIX directory object). The set of important related inodes are:

- **Volume info inode:** This field identifies the inode which hosts the on-disk representation of the volume's header. It is very similar to the information pointed to by the `volume` field of the `struct volser_trans` defined in Section 5.4.1, recording important status information for the volume.
- **Large vnode index inode:** This field identifies the inode which holds the list of vnode identifiers for all directory objects residing within the volume. These are "large" since they must also hold the Access Control List (ACL) information for the given AFS directory.
- **Small vnode index inode:** This field identifies the inode which holds the list of vnode identifiers for all non-directory objects hosted by the volume.

All of the actual files and directories residing within an AFS volume, as identified by the contents of the large and small vnode index inodes, are also free-floating inodes, not appearing in the conventional UNIX name space. This is the reason the vendor-supplied `fsck` program should **not** be run on partitions containing AFS volumes. Since the inodes making up AFS files and directories, as well as the inodes serving as volume indices for them, are not mapped to any directory, the standard `fsck` program would throw away all of these "unreferenced" inodes. Thus, a special version of `fsck` is provided that recognizes partitions containing AFS volumes as well as standard UNIX partitions.

4.3 Transactions

Each individual volume operation is carried out by the *Volume Server* as a *transaction*, but not in the atomic sense of the word. Logically, creating a *Volume Server* transaction can be equated with performing an "exclusive open" on the given volume before beginning the actual work of the desired volume operation. No other *Volume Server* (or *File Server*) operation is allowed on the opened volume until the transaction is terminated. Thus, transactions in the context of the *Volume Server* serve to provide mutual exclusion without any of the normal atomicity guarantees. Volumes maintain enough internal state to enable recovery from interrupted or failed operations via use of the `salvager` program. Whenever volume inconsistencies are detected, this `salvager` program is run, which then attempts to correct the problem.

Volume transactions have timeouts associated with them. This guarantees that the death of the agent performing a given volume operation cannot result in the volume being permanently removed from circulation. There are actually two timeout periods defined for a volume transaction. The first is the *warning time*, defined to be 5 minutes. If a transaction lasts for more than this time period without making progress, the *Volume Server* prints a warning message to its log file (see Section 4.5). The second time value associated with a volume transaction is the *hard timeout*, defined to occur 10 minutes after any progress has been made on the given operation. After this period, the transaction will be unconditionally deleted, and the volume freed for any other operations. Transactions are reference-counted. Progress will be deemed to have occurred for a transaction, and its internal timeclock field will be updated, when:

1. The transaction is first created.
2. A reference is made to the transaction, causing the *Volume Server* to look it up in its internal tables.
3. The transaction's reference count is decremented.

4.4 The *volserver* Process

The *volserver* user-level program is run on every AFS server machine, and implements the *Volume Server* agent. It is responsible for providing the *Volume Server* interface as defined by the *volint.xg Rxgen* file.

The *volserver* process defines and launches five threads to perform the bulk of its duties. One thread implements a background daemon whose job it is to garbage-collect timed-out transaction structures. The other four threads are RPC interface listeners, primed to accept remote procedure calls and thus perform the defined set of volume operations.

Certain non-standard configuration settings are made for the RPC subsystem by the *volserver* program. For example, it chooses to extend the length of time that an *Rx* connection may remain idle from the default 12 seconds to 120 seconds. The reasoning here is that certain volume operations may take longer than 12 seconds of processing time on the server, and thus the default setting for the connection timeout value would incorrectly terminate an RPC when in fact it was proceeding normally and correctly.

The *volserver* program takes a single, optional command line argument. If a positive integer value is provided on the command line, then it shall be used to set the debugging level within the *Volume Server*. By default, a value of zero is used, specifying that no special debugging output will be generated and fed to the *Volume Server* log file described below.

4.5 Log File

The *Volume Server* keeps a log file, recording the set of events of special interest it has encountered. The file is named `VolserLog`, and is stored in the */usr/afs/logs* directory on the local disk of the server machine on which the *Volume Server* runs. This is a human-readable file, with every entry time-stamped.

Whenever the *volserver* program restarts, it renames the current *VolserLog* file to *VolserLog.old*, and starts up a fresh log. A properly-authorized individual can easily inspect the log file residing on any given server machine. This is made possible by the *BOS Server* AFS agent running on the machine, which allows the contents of this file to be fetched and displayed on the caller's machine via the `bos getlog` command.

An excerpt from a *Volume Server* log file follows below. The numbers appearing in square brackets at the beginning of each line have been inserted so that we may reference the individual lines of the log excerpt in the following paragraph.

```
[1] Wed May  8 06:03:00 1991 AttachVolume: Error attaching volume
        /vicepd/V1969547815.vol; volume needs salvage
[2] Wed May  8 06:03:01 1991 Volser: ListVolumes: Could not attach volume 1969547815
[3] Wed May  8 07:36:13 1991 Volser: Clone: Cloning volume 1969541499 to new
        volume 1969541501
[4] Wed May  8 11:25:05 1991 AttachVolume: Cannot read volume header
        /vicepd/V1969547415.vol
[5] Wed May  8 11:25:06 1991 Volser: CreateVolume: volume 1969547415
        (bld.dce.s3.dv.pmax_ul3) created
```

Line [1] indicates that the volume whose numerical ID is 1969547815 could not be attached on partition `/vicepd`. This error is probably the result of an aborted transaction which left the volume in an inconsistent state, or by actual damage to the volume structure or data. In this case, the *Volume Server* recommends that the `salvager` program be run on this volume to restore its integrity. Line [2] records the operation which revealed this situation, namely the invocation of an *AFSVolListVolumes()* RPC.

Line [4] reveals that the volume header file for a specific volume could not be read. Line [5], as with line [2] in the above paragraph, indicates why this is true. Someone had called the *AFSVolCreateVolume()* interface function, and as a precaution, the *Volume Server* first checked to see if such a volume was already present by attempting to read its header.

Thus verifying that the volume did not previously exist, the *Volume Server* allowed the *AFSVolCreateVolume()* call to continue its processing, creating and initializing the proper volume file, *V1969547415.vol*, and the associated header and index inodes.

Chapter 5

Volume Server **Interface**

5.1 Introduction

This chapter documents the API for the *Volume Server* facility, as defined by the *volint.xg Rxgen* interface file and the *volser.h* include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer appear here.

5.2 Constants

This section covers the basic constant definitions of interest to the *Volume Server* application programmer. These definitions appear in the *volint.h* file, automatically generated from the *volint.xg Rxgen* interface file, and in *volser.h*.

Each subsection is devoted to describing the constants falling into the following categories:

- Configuration and boundary values
- Interface routine opcodes
- Transaction Flags
- Volume Types
- LWP State

- States for `struct vldbentry`
- Validity Checks
- Miscellaneous

5.2.1 Configuration and Boundary Values

These constants define some basic system configuration values, along with such things as maximum sizes of important arrays.

Name	*Value*	*Description*
`MyPort`	5,003	The *Rx* UDP port on which the *Volume Server* service may be found.
`NameLen`	80	Used by the `vos` utility to define maximum lengths for internal filename variables.
`VLDB_MAXSERVERS`	10	Maximum number of server agents implementing the AFS Volume Location Database (VLDB) for the cell.
`VOLSERVICE_ID`	4	The *Rx* service number on the given UDP port (`MyPort`) above.
`INVALID_BID`	0	Used as an invalid read-only or backup volume ID.
`VOLSER_MAXVOLNAME`	65	The number of characters in the longest possible volume name, including the trailing null. **Note:** this is only used by the `vos` utility; the *Volume Server* uses the "old" value below.
`VOLSER_OLDMAXVOLNAME`	32	The "old" maximum number of characters in an AFS volume name, including the trailing null. In reality, it is also the current maximum.
`VOLSER_MAX_REPSITES`	7	The maximum number of replication sites for a volume.
`VNAMESIZE`	32	Size in bytes of the `name` field in `struct volintInfo` (see Section 5.4.6).

5.2.2 Interface Routine Opcodes

These constants, appearing in the *volint.xg Rxgen* interface file for the *Volume Server*, define the opcodes for the RPC routines. Every *Rx* call on this interface contains this opcode, and the dispatcher uses it to select the proper code at the server site to carry out the call.

Name	*Value*	*Description*
VOLCREATEVOLUME	100	Opcode for *AFSVolCreateVolume()*
VOLDELETEVOLUME	101	Opcode for *AFSVolDeleteVolume()*
VOLRESTORE	102	Opcode for *AFSVolRestoreVolume()*
VOLFORWARD	103	Opcode for *AFSVolForward()*
VOLENDTRANS	104	Opcode for *AFSVolEndTrans()*
VOLCLONE	105	Opcode for *AFSVolClone()* .
VOLSETFLAGS	106	Opcode for *AFSVolSetFlags()*
VOLGETFLAGS	107	Opcode for *AFSVolGetFlags()*
VOLTRANSCREATE	108	Opcode for *AFSVolTransCreate()*
VOLDUMP	109	Opcode for *AFSVolDump()*
VOLGETNTHVOLUME	110	Opcode for *AFSVolGetNthVolume()*
VOLSETFORWARDING	111	Opcode for *AFSVolSetForwarding()*
VOLGETNAME	112	Opcode for *AFSVolGetName()*
VOLGETSTATUS	113	Opcode for *AFSVolGetStatus()*
VOLSIGRESTORE	114	Opcode for *AFSVolSignalRestore()*
VOLLISTPARTITIONS	115	Opcode for *AFSVolListPartitions()*
VOLLISTVOLS	116	Opcode for *AFSVolListVolumes()*
VOLSETIDSTYPES	117	Opcode for *AFSVolSetIdsTypes()*
VOLMONITOR	118	Opcode for *AFSVolMonitor()*
VOLDISKPART	119	Opcode for *AFSVolPartitionInfo()*
VOLRECLONE	120	Opcode for *AFSVolReClone()*
VOLLISTONEVOL	121	Opcode for *AFSVolListOneVolume()*
VOLNUKE	122	Opcode for *AFSVolNukeVolume()*
VOLSETDATE	123	Opcode for *AFSVolSetDate()*

5.2.3 Transaction Flags

These constants define the various flags the *Volume Server* uses in assocation with volume transactions, keeping track of volumes upon which operations are currently proceeding. There are three sets of flag values, stored in three different fields within a `struct volser_trans`: general volume state, attachment modes, and specific transaction states.

5.2.3.1 vflags

These values are used to represent the general state of the associated volume. They appear in the `vflags` field within a `struct volser_trans`.

Name	*Value*	*Description*
`VTDeleteOnSalvage`	1	The volume should be deleted on next salvage.
`VTOutOfService`	2	This volume should never be put online.
`VTDeleted`	4	This volume has been deleted (via *AFSVolDeleteVolume()*), and thus should not be manipulated.

5.2.3.2 iflags

These constants represent the desired *attachment mode* for a volume at the start of a transaction. Once attached, the volume header is marked to reflect this mode. Attachment modes are useful in salvaging partitions, as they indicate whether the operations being performed on individual volumes at the time the crash occured could have introduced inconsistencies in their metadata descriptors. If a volume was attached in a read-only fashion, then the salvager may decide (taking other factors into consideration) that the volume doesn't need attention as a result of the crash.

These values appear in the `iflags` field within a `struct volser_trans`.

Name	*Value*	*Description*
`ITOffline`	0x1	Volume offline on server (returns `VOFFLINE`).
`ITBusy`	0x2	Volume busy on server (returns `VBUSY`).
`ITReadOnly`	0x8	Volume is read-only on client, read-write on server - DO NOT USE.
`ITCreate`	0x10	Volume does not exist correctly yet.
`ITCreateVolID`	0x1000	Create volid.

5.2.3.3 tflags

This value is used to represent the transaction state of the associated volume, and appears in the `tflags` field within a `struct volser_trans`.

Name	*Value*	*Description*
`TTDeleted`	1	Delete transaction not yet freed due to high reference count.

5.2.4 Volume Types

The following constants may be supplied as values for the `type` argument to the *AFSVolCreateVolume()* interface call. They are just synonyms for the three values

RWVOL, ROVOL, and BACKVOL.

Name	*Value*	*Description*
volser_RW	0	Specifies a read-write volume type.
volser_RO	1	Specifies a read-only volume type.
volser_BACK	2	Specifies a backup volume type.

5.2.5 LWP State

This set of exported definitions refers to objects internal to the *Volume Server*, and strictly speaking should not be visible to other agents. Specifically, a busyFlags array keeps a set of flags referenced by the set of lightweight threads running within the *Volume Server*. These flags reflect and drive the state of each of these worker LWPs.

Name	*Value*	*Description*
VHIdle	1	*Volume Server* LWP is idle, waiting for new work.
VHRequest	2	A work item has been queued.

5.2.6 States for struct vldbentry

The *Volume Server* defines a collection of synonyms for certain values defined by the *Volume Location Server*. These particular constants are used within the flags field in objects of type struct vldbentry. The equivalent *Volume Location Server* values are described in Section 3.2.6.

Name	*Value*	*Description*
RW_EXISTS	0x1000	Synonym for VLF_RWEXISTS.
RO_EXISTS	0x2000	Synonym for VLF_ROEXISTS.
BACK_EXISTS	0x4000	Synonym for VLF_BACKEXISTS.
NEW_REPSITE	0x01	Synonym for VLSF_NEWREPSITE.
ITSROVOL	0x02	Synonym for VLFS_ROVOL.
ITSRWVOL	0x04	Synonym for VLSF_RWVOL.
ITSBACKVOL	0x08	Synonym for VLSF_BACKVOL.

5.2.7 Validity Checks

These values are used for performing validity checks. The first one appears only within the partFlags field within objects of type partList (see Section 5.4.3). The rest (except

VOK and VBUSY) appear in the volFlags field within an object of type struct volDescription. These latter defintions are used within the volFlags field to mark whether the rest of the fields within the struct volDescription are valid. Note that while several constants are defined, only some are actually used internally by the *Volume Server* code.

Name	*Value*	*Description*
PARTVALID	0x01	The indicated partition is valid.
CLONEVALID	0x02	The indicated clone (field volCloneId) is a valid one.
CLONEZAPPED	0x04	The indicated clone volume (field volCloneId) has been deleted.
IDVALID	0x08	The indicated volume ID (field volId) is valid.
NAMEVALID	0x10	The indicted volume name (field volName) is valid. Not used internally by the *Volume Server*.
SIZEVALID	0x20	The indicated volume size (field volSize) is valid. Not used internally by the *Volume Server*.
ENTRYVALID	0x40	The struct volDescription refers to a valid volume.
REUSECLONEID	0x80	The indicated clone ID (field volCloneId) should be reused.
VOK	0x02	Used in the status field of struct volintInfo to show that everything is OK.
VBUSY	110	Used in the status field of struct volintInfo to show that the volume is currently busy.

5.2.8 Miscellaneous

This section covers the set of exported *Volume Server* definitions that don't easily fall into the above categories.

Name	*Value*	*Description*
SIZE	1,024	Not used internally by the *Volume Server*; used as a maximum size for internal character arrays.
MAXHELPERS	10	Size of an internal *Volume Server* character array (busyFlags), it marks the maximum number of threads within the server.
STDERR	stderr	Synonym for the UNIX standard input file descriptor.
STDOUT	stdout	Synonym for the UNIX standard output file descriptor.

5.3 Exported Variables

This section describes the single variable that the *Volume Server* exports to its applications.

The `QI_GlobalWriteTrans` exported variable represents a pointer to the head of the global queue of transaction structures for operations being handled by a *Volume Server*. Each object in this list is of type `struct volser_trans` (see Section 5.4.1 below).

5.4 Structures and Typedefs

This section describes the major exported *Volume Server* data structures of interest to application programmers, along with some of the the typedefs based on those structures. Please note that typedefs in shose definitions angle brackets appear are those fed through the *Rxgen* RPC stub generator. *Rxgen* uses these angle brackets to specify an array of indefinite size.

5.4.1 `struct volser_trans`

This structure defines the transaction record for all volumes upon which an active operation is proceeding.

Fields

struct volser_trans *next - Pointer to the next transaction structure in the queue.

long tid - Transaction ID.

long time - The time this transaction was last active, for timeout purposes. This is the standard UNIX time format.

long creationTime - The time a which this transaction started.

long returnCode - The overall transaction error code.

struct Volume *volume - Pointer to the low-level object describing the associated volume. This is included here for the use of lower-level support code.

long volid - The associated volume's numerical ID.

long partition - The partition on which the given volume resides.

long dumpTransId - Not used.

long dumpSeq - Not used.

short refCount - Reference count on this structure.

short iflags - Initial attach mode flags.

char vflags - Current volume status flags.

char tflags - Transaction flags.

char incremental - If non-zero, indicates that an incremental restore operation should be performed.

char lastProcName[] - Name of the last internal *Volume Server* procedure that used this transaction. This field may be up to 30 characters long, including the trailing null, and is intended for debugging purposes only.

struct rx_call *rxCallPtr - Pointer to latest associated rx_call. This field is intended for debugging purposes only.

5.4.2 struct volDescription

This structure is used by the AFS backup system to group certain key fields of volume information.

Fields

char volName[] - The name of the given volume; maximum length of this string is `VOLSER_MAXVOLNAME` characters, including the trailing null.

long volId - The volume's numerical ID.

int volSize - The size of the volume, in bytes.

long volFlags - Keeps validity information on the given volume and its clones. This field takes on values from the set defined in Section 5.2.7

long volCloneId - The volume's current clone ID.

5.4.3 struct partList

This structure is used by the backup system and the `vos` tool to keep track of the state of the AFS disk partitions on a given server.

Fields

long partId[] - Set of 26 partition IDs.

long partFlags[] - Set to `PARTVALID` if the associated partition slot corresponds to a valid partition. There are 26 entries in this array.

5.4.4 struct volser_status

This structure holds the status of a volume as it is known to the *Volume Server*, and is passed to clients through the *AFSVolGetStatus()* interface call.

Two fields appearing in this structure, `accessDate` and `updateDate`, deserve a special note. In particular, it is important to observe that these fields are **not** kept in full synchrony with reality. When a *File Server* provides one of its client *Cache Managers* with a chunk of a file on which to operate, it is incapable of determining *exactly* when the data in that chunk is accessed, or *exactly* when it is updated. This is because the manipulations occur on the client machine, without any information on these accesses or updates passed back to the server. The only time these fields can be modified is when the chunk of a file resident within the given volume is delivered to a client (in the case of `accessDate`), or when a client writes back a dirty chunk to the *File Server* (in the case of `updateDate`).

Fields

long volID - The volume's numerical ID, unique within the cell.

long nextUnique - Next value to use for a vnode uniquifier within this volume.

int type - Basic volume class, one of `RWVOL`, `ROVOL`, or `BACKVOL`.

long parentID - Volume ID of the parent, if this volume is of `type` `ROVOL` or `BACKVOL`.

long cloneID - ID of the latest read-only clone, valid iff the `type` field is set to `RWVOL`.

long backupID - Volume ID of the latest backup of this read-write volume.

long restoredFromID - The volume ID contained in the dump from which this volume was restored. This field is used to simply make sure that an incremental dump is not restored on top of something inappropriate. Note that this field itself is not dumped.

long maxQuota - The volume's maximum quota, in 1Kbyte blocks.

long minQuota - The volume's minimum quota, in 1Kbyte blocks.

long owner - The user ID of the person responsible for this volume.

long creationDate - For a volume of type `RWVOL`, this field marks its creation date. For the original copy of a clone, this field represents the cloning date.

long accessDate - Last access time by a user for this volume. This value is expressed as a standard UNIX longword date quantity.

long updateDate - Last modification time by a user for this volume. This value is expressed as a standard UNIX longword date quantity.

long expirationDate - Expiration date for this volume. If the volume never expires, then this field is set to zero.

long backupDate - The last time a backup clone was created for this volume.

long copyDate - The time that this copy of this volume was created.

5.4.5 `struct destServer`

Used to specify the destination server in an *AFSVolForward()* invocation (see Section 5.7.7).

Fields

long destHost - The IP address of the destination server.

long destPort - The UDP port for the *Volume Server Rx* service there.

long destSSID - Currently, this field is always set to 1.

5.4.6 `struct volintInfo`

This structure is used to communicate volume information to the *Volume Server*'s RPC clients. It is used to build the `volEntries` object, which appears as a parameter to the *AFSVolListVolumes()* call.

The comments in Section 5.4.4 concerning the `accessDate` and `updateDate` fields are equally valid for the analogue fields in this structure.

Fields

char name[] - The null-terminated name for the volume, which can be no longer than `VNAMESIZE` (32) characters, including the trailing null.

long volid - The volume's numerical ID.

long type - The volume's basic class, one of `RWVOL`, `ROVOL`, or `BACKVOL`.

long backupID - The latest backup volume's ID.

long parentID - The parent volume's ID.

long cloneID - The latest clone volume's ID.

long status - Status of the volume; may be one of `VOK` or `VBUSY`.

long copyDate - The time that this copy of this volume was created.

unsigned char inUse - If non-zero, an indication that this volume is online.

unsigned char needsSalvaged - If non-zero, an indication that this volume needs to be salvaged.

unsigned char destroyMe - If non-zero, an indication that this volume should be destroyed.

long creationDate - Creation date for a read/write volume; cloning date for the original copy of a read-only volume.

long accessDate - Last access time by a user for this volume.

long updateDate - Last modification time by a user for this volume.

long backupDate - Last time a backup copy was made of this volume.

int dayUse - Number of times this volume was accessed since midnight of the current day.

int filecount - the number of file system objects contained within the volume.

int maxquota - The upper limit on the number of 1-Kbyte disk blocks of storage that this volume may obtain.

int size - Not known.

long flags - Values used by the backup system are stored here.

long spare1 - spare3 - Spare fields, reserved for future use.

5.4.7 struct transDebugInfo

This structure is provided for monitoring and debugging purposes. It is used to compose the `transDebugEntries` variable-sized object, which in turn appears as a parameter to the *AFSVolMonitor()* interface call.

Fields

long tid - The transaction ID.

long time - The time when the transaction was last active, for timeout purposes.

long creationTime - The time the transaction started.

long returnCode - The overall transaction error code.

long volid - The open volume's ID.

long partition - The open volume's partition.

short iflags - Initial attach mode flags (IT*).

char vflags - Current volume status flags (VT*).

char tflags - Transaction flags (TT*).

char lastProcName[] - The string name of the last procedure which used transaction. This field may be up to 30 characters long, including the trailing null, and is intended for debugging purposes only.

int callValid - Flag which determines if the following fields are valid.

long readNext - Sequence number of the next *Rx* packet to be read.

long transmitNext - Sequence number of the next *Rx* packet to be transmitted.

int lastSendTime - The last time anything was sent over the wire for this transaction.

int lastReceiveTime - The last time anything was received over the wire for this transaction.

5.4.8 `struct pIDs`

Used by the *AFSVolListPartitions()* interface call, this structure is used to store information on all of the partitions on a given *Volume Server*.

Fields

long partIds[] - One per letter of the alphabet (/vicepa through /vicepz). Filled with 0 for '/vicepa', 25 for '/vicepz'. Invalid partition slots are filled in with a -1.

5.4.9 `struct diskPartition`

This structure contains information regarding an individual AFS disk partition. It is returned as a parameter to the *AFSVolPartitionInfo()* call.

Fields

- **char name[]** - Mounted partition name, up to 32 characters long including the trailing null.
- **char devName[]** - Device name on which the partition lives, up to 32 characters long including the trailing null.
- **int lock_fd** - A lock used for mutual exclusion to the named partition. A value of -1 indicates the lock is not currently being held. Otherwise, it has the file descriptor resulting from the `UNIX` *open()* call on the file specified in the `name` field used to "acquire" the lock.
- **int totalUsable** - The number of blocks within the partition which are available.
- **int free** - The number of free blocks in the partition.
- **int minFree** - The minimum number of blocks that must remain free regardless of allocation requests.

5.4.10 `struct restoreCookie`

Used as a parameter to both *AFSVolRestore()* and *AFSVolForward()*, a `restoreCookie` keeps information that must be preserved between various *Volume Server* operations.

Fields

- **char name[]** - The volume name, up to 32 characters long including the trailing null.
- **long type** - The volume type, one of `RWVOL`, `ROVOL`, and `BACKVOL`.
- **long clone** - The current read-only clone ID for this volume.
- **long parent** - The parent ID for this volume.

5.4.11 `transDebugEntries`

```
typedef transDebugInfo transDebugEntries<>;
```

This typedef is used to generate a variable-length object which is passed as a parameter to the *AFSVolMonitor()* interface function. Thus, it may carry any number of descriptors for active transactions on the given *Volume Server*. Specifically, it causes a C structure of the same name to be defined with the following fields:

Fields

u_int transDebugEntries_len - The number of `struct transDebugInfo` (see Section 5.4.7) objects appearing at the memory location pointed to by the `transDebugEntries_val` field.

transDebugInfo *transDebugEntries_val - A pointer to a region of memory containing an array of `transDebugEntries_len` objects of type `struct transDebugInfo`.

5.4.12 volEntries

```
typedef volintInfo volEntries<>;
```

This typedef is used to generate a variable-length object which is passed as a parameter to *AFSVolListVolumes()*. Thus, it may carry any number of descriptors for volumes on the given *Volume Server*. Specifically, it causes a C structure of the same name to be defined with the following fields:

Fields

u_int volEntries_len - The number of `struct volintInfo` (see Section 5.4.6) objects appearing at the memory location pointed to by the `volEntries_val` field.

volintInfo *volEntries_val - A pointer to a region of memory containing an array of `volEntries_len` objects of type `struct volintInfo`.

5.5 Error Codes

The *Volume Server* advertises two groups of error codes. The first set consists of the standard error codes defined by the package. The second is a collection of lower-level return values which are exported here for convenience.

5.5.1 Standard

The error codes described in this section were defined by the *Volume Server* to describe exceptional conditions arising in the course of RPC call handling.

Name	*Value*	*Description*
VOLSERTRELE_ERROR	1492325120L	internal error releasing transaction.
VOLSERNO_OP	1492325121L	unknown internal error.
VOLSERREAD_DUMPERROR	1492325122L	badly formatted dump.
VOLSERDUMPERROR	1492325123L	badly formatted dump(2).
VOLSERATTACH_ERROR	1492325124L	could not attach volume.
VOLSERILLEGAL_PARTITION	1492325125L	illegal partition.
VOLSERDETACH_ERROR	1492325126L	could not detach volume.
VOLSERBAD_ACCESS	1492325127L	insufficient privilege for volume operation.
VOLSERVLDB_ERROR	1492325128L	error from volume location database.
VOLSERBADNAME	1492325129L	bad volume name.
VOLSERVOLMOVED	1492325130L	volume moved.
VOLSERBADOP	1492325131L	illegal volume operation.
VOLSERBADRELEASE	1492325132L	volume release failed.
VOLSERVOLBUSY	1492325133L	volume still in use by volserver.
VOLSERNO_MEMORY	1492325134L	out of virtual memory in volserver.
VOLSERNOVOL	1492325135L	no such volume.
VOLSERMULTIRWVOL	1492325136L	more than one read/write volume.
VOLSERFAILEDOP	1492325137L	failed volume server operation.

5.5.2 Low-Level

These error codes are duplicates of those defined from a package which is internal to the *Volume Server*. They are re-defined here to make them visible to *Volume Server* clients.

Name	*Value*	*Description*
VSALVAGE	101	Volume needs to be salvaged.
VNOVNODE	102	Bad vnode number encountered.
VNOVOL	103	The given volume is either not attached, doesn't exist, or is not online.
VVOLEXISTS	104	The given volume already exists.
VNOSERVICE	105	The volume is currently not in service.
VOFFLINE	106	The specified volume is offline, for the reason given in the offline message field (a subfield within the `volume` field in `struct volser_trans`).
VONLINE	107	Volume is already online.
VDISKFULL	108	The disk partition is full.
VOVERQUOTA	109	The given volume's maximum quota, as expressed in the `maxQuota` field of the `struct volintInfo`, has been exceeded.
VBUSY	110	The named volume is temporarily unavailable, and the client is encouraged to retry the operation shortly.
VMOVED	111	The given volume has moved to a new server.

The `VICE_SPECIAL_ERRORS` constant is defined to be the lowest of these error codes.

5.6 Macros

The *Volume Server* defines a small number of macros, as described in this section.

5.6.1 *THOLD()*

```
#define THOLD(tt)  ((tt)->refCount++)
```

This macro is used to increment the reference count field, `refCount`, in an object of type `struct volser_trans`. Thus, the associated transaction is effectively "held", insuring it won't be garbage-collected. The counterpart to this operation, `TRELE()`, is implemented by the *Volume Server* as a function.

5.6.2 *ISNAMEVALID()*

```
#define ISNAMEVALID(name) (strlen(name) < (VOLSER_OLDMAXVOLNAME - 9))
```

This macro checks to see if the given `name` argument is of legal length. It must be no more than the size of the container, which is at most `VOLSER_OLDMAXVOLNAME` characters, minus the length of the longest standardized volume name postfix known to the system. That postfix is the 9-character `.restored` string, which is tacked on to the name of a volume that has been restored from a dump.

5.7 Functions

This section covers the *Volume Server* RPC interface routines, defined by and generated from the *volint.xg Rxgen* file. The following is a summary of the interface functions and their purpose:

Fcn Name	*Description*
`AFSVolCreateVolume`	Create a volume.
`AFSVolDeleteVolume`	Delete a volume.
`AFSVolNukeVolume`	Obliterate a volume completely.
`AFSVolDump`	Dump (i.e., save) the contents of a volume.
`AFSVolSignalRestore`	Show intention to call *AFSVolRestore()*.
`AFSVolRestore`	Recreate a volume from a dump.
`AFSVolForward`	Dump a volume, then restore to a given server and volume.
`AFSVolClone`	Clone (and optionally purge) a volume.
`AFSVolReClone`	Re-clone a volume.
`AFSVolSetForwarding`	Set forwarding info for a moved volume.
`AFSVolTransCreate`	Create transaction for a [volume, partition].
`AFSVolEndTrans`	End a transaction.
`AFSVolGetFlags`	Get volume flags for a transaction.
`AFSVolSetFlags`	Set volume flags for a transaction.
`AFSVolGetName`	Get the volume name associated with a transaction.
`AFSVolGetStatus`	Get status of a transaction/volume.
`AFSVolSetIdsTypes`	Set header info for a volume.
`AFSVolSetDate`	Set creation date in a volume.
`AFSVolListPartitions`	Return a list of AFS partitions on a server.
`AFSVolPartitionInfo`	Get partition information.
`AFSVolListVolumes`	Return a list of volumes on the server.
`AFSVolListOneVolume`	Return header info for a single volume.
`AFSVolGetNthVolume`	Get volume header given its index.
`AFSVolMonitor`	Collect server transaction state.

There are two general comments that apply to most of the *Volume Server* interface routines:

1. AFS partitions are identified by integers ranging from 0 to 25, corresponding to the letters 'a' through 'z'. By convention, AFS partitions are named `/vicepx`, where x is any lower-case letter.

2. Legal volume types to pass as parameters are `RWVOL`, `ROVOL`, and `BACKVOL`, as defined in Section 3.2.4.

5.7.1 AFSVolCreateVolume — Create a volume

```
int AFSVolCreateVolume(IN struct rx_connection *z_conn,
                       IN long partition,
                       IN char *name,
                       IN long type,
                       IN long parent,
                       INOUT long *volid,
                       OUT long *trans)
```

Description

Create a volume named `name`, with numerical identifier `volid`, and of type `type`. The new volume is to be placed in the specified `partition` for the server machine as identified by the *Rx* connection information pointed to by `z_conn`. If a value of 0 is provided for the `parent` argument, it will be set by the *Volume Server* to the value of `volid` itself. The `trans` parameter is set to the *Volume Location Server* transaction ID corresponding to the volume created by this call, if successful.

The numerical volume identifier supplied in the `volid` parameter must be generated beforehand by calling *VL_GetNewVolumeID()* (see Section 3.6.5). After *AFSVolCreateVolume()* completes correctly, the new volume is marked as offline. It must be explicitly brought online through a call to *AFSVolSetFlags()* (see Section 5.7.14) while passing the `trans` transaction ID generated by `AFSVolCreateVolume()`. The "hold" on the new volume guaranteed by the `trans` transaction may be "released" by calling *AFSVolEndTrans()*. Until then, no other process may operate on the volume.

Upon creation, a volume's maximum quota (as specified in the `maxquota` field of a `struct volintInfo`) is set to 5,000 1-Kbyte blocks.

Note that the *AFSVolCreateVolume()* routine is the only *Volume Server* function that manufactures its own transaction. All others must have already acquired a transaction ID via either a previous call to *AFSVolCreateVolume()* or *AFSVolTransCreate()*.

Error Codes

`VOLSERBADNAME` The volume `name` parameter was longer than 31 characters plus the trailing null.

`VOLSERBAD_ACCESS` The caller is not authorized to create a volume.

`EINVAL` The `type` parameter was illegal.

`E2BIG` A value of `0` was provided in the `volid` parameter.

`VOLSERVOLBUSY` A transaction could not be created, thus the given volume was busy.

`EIO` The new volume entry could not be created.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

`<misc>` If the `partition` parameter is unintelligible, this routine will return a low-level `UNIX` error.

5.7.2 AFSVolDeleteVolume — Delete a volume

```
int AFSVolDeleteVolume(IN struct rx_connection *z_conn,
                       IN long trans)
```

Description

Delete the volume associated with the open transaction ID specified within `trans`. All of the file system objects contained within the given volume are destroyed, and the on-disk volume metadata structures are reclaimed. In addition, the in-memory volume descriptor's `vflags` field is set to `VTDeleted`, indicating that it has been deleted.

Under some circumstances, a volume should be deleted by calling *AFSVolNukeVolume()* instead of this routine. See Section 5.7.3 for more details.

Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to delete a volume.

`ENOENT` The `trans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

5.7.3 AFSVolNukeVolume — Obliterate a volume completely

```
int AFSVolNukeVolume(IN struct rx_connection *z_conn,
                     IN long partID,
                     IN long volID)
```

Description

Completely obliterate the volume living on partition `partID` whose ID is `volID`. This involves scanning all inodes on the given partition and removing those marked with the specified `volID`. If the volume is a read-only clone, only the header inodes are removed, since they are the only ones stamped with the read-only ID. To reclaim the space taken up by the actual data referenced through a read-only clone, this routine should be called on the read-write master. Note that calling *AFSVolNukeVolume()* on a read-write volume effectively destroys all the read-only volumes cloned from it, since everything except for their indicies to the (now-deleted) data will be gone.

Under normal circumstances, it is preferable to use *AFSVolDeleteVolume()* instead of *AFSVolNukeVolume()* to delete a volume. The former is much more efficient, as it only touches those objects in the partition that belong to the named volume, walking the on-disk volume metadata structures. However, *AFSVolNukeVolume()* **must** be used in situations where the volume metadata structures are known to be damaged. Since a complete scan of all inodes in the partition is performed, all disconnected or unreferenced portions of the given volume will be reclaimed.

Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to call this routine.

`VOLSERNOVOL` The partition specified by the `partID` argument is illegal.

5.7.4 AFSVolDump — Dump (i.e., save) the contents of a volume

```
int AFSVolDump(IN struct rx_connection *z_conn,
               IN long fromTrans,
               IN long fromDate)
```

Description

Generate a canonical dump of the contents of the volume associated with transaction `fromTrans` as of calendar time `fromDate`. If the given `fromDate` is zero, then a full dump will be carried out. Otherwise, the resulting dump will be an incremental one.

This is specified as a *split* function within the *volint.xg Rxgen* interface file. This specifies that *two* routines are generated, namely *StartAFSVolDump()* and *EndAFSVolDump()*. The former is used to marshall the `IN` arguments, and the latter is used to unmarshall the return value of the overall operation. The actual dump data appears in the *Rx* stream for the call (see the section entitled *Example Server and Client* in the companion AFS-3 Programmer's Reference: Specification for the *Rx* Remote Procedure Call Facility document).

Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to dump a volume.

`ENOENT` The `fromTrans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

5.7.5 AFSVolSignalRestore — Show intention to call *AFSVolRestore()*

```
int AFSVolSignalRestore(IN struct rx_connection *z_conn,
                        IN char *name,
                        IN int type,
                        IN long pid,
                        IN long cloneid)
```

Description

Show an intention to the *Volume Server* that the client will soon call *AFSVolRestore()*. The parameters, namely the volume `name`, `type`, parent ID `pid` and clone ID `cloneid` are stored in a well-known set of global variables. These values are used to set the restored volume's header, overriding those values present in the dump from which the volume will be resurrected.

Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to call this routine.

`VOLSERBADNAME` The volume name contained in `name` was longer than 31 characters plus the trailing null.

5.7.6 AFSVolRestore — Recreate a volume from a dump

```
int AFSVolRestore(IN struct rx_connection *z_conn,
                  IN long toTrans,
                  IN long flags,
                  IN struct restoreCookie *cookie)
```

Description

Interpret a canonical volume dump (generated as the result of calling *AFSVolDumpVolume()*), passing it to the volume specified by the `toTrans` transaction. Only the low bit in the `flags` argument is inspected. If this low bit is turned on, the dump will be restored as incremental; otherwise, a full restore will be carried out.

All callbacks to the restored volume are broken.

This is specified as a *split* function within the *volint.xg Rxgen* interface file. This specifies that *two* routines are generated, namely *StartAFSVolRestore()* and *EndAFSVolRestore()* . The former is used to marshall the `IN` arguments, and the latter is used to unmarshall the return value of the overall operation. The actual dump data flows over the *Rx* stream for the call (see the section entitled *Example Server and Client* in the companion AFS-3 Programmer's Reference: Specification for the *Rx* Remote Procedure Call Facility document).

The *AFSVolSignalRestore()* routine (see Section 5.7.5) should be called before invoking this function in order to signal the intention to restore a particular volume.

Error Codes

`VOLSERREAD_DUMPERROR` Dump data being restored is corrupt.

`VOLSERBAD_ACCESS` The caller is not authorized to restore a volume.

`ENOENT` The `fromTrans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

5.7.7 AFSVolForward — Dump a volume, then restore to given server and volume

```
int AFSVolForward(IN struct rx_connection *z_conn,
                  IN long fromTrans,
                  IN long fromDate,
                  IN struct destServer *destination,
                  IN long destTrans,
                  IN struct restoreCookie *cookie)
```

Description

Dumps the volume associated with transaction `fromTrans` from the given `fromDate`. The dump itself is sent to the server described by `destination`, where it is restored as the volume associated with transaction `destTrans`. In reality, an *Rx* connection is set up to the `destServer`, *StartAFSVolRestore()* directs writing to the *Rx* call's stream, and then *EndAFSVolRestore()* is used to deliver the dump for the volume corresponding to `fromTrans`. If a non-zero `fromDate` is provided, then the dump will be incremental from that date. Otherwise, a full dump will be delivered.

The *Rx* connection set up for this task is always destroyed before the function returns. The destination volume should exist before carrying out this operation, and the invoking process should have started transactions on both participating volumes.

Error Codes

- `VOLSERBAD_ACCESS` The caller is not authorized to forward a volume.
- `ENOENT` The `fromTrans` transaction was not found.
- `VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.
- `ENOTCONN` An *Rx* connection to the destination server could not be established.

5.7.8 AFSVolClone — Clone (and optionally purge) a volume

```
int AFSVolClone(IN struct rx_connection *z_conn,
                IN long trans,
                IN long purgeVol,
                IN long newType,
                IN char *newName,
                INOUT long *newVol)
```

Description

Make a clone of the read-write volume associated with transaction `trans`, giving the cloned volume a name of `newName`. The `newType` parameter specifies the type for the new clone, and may be either `ROVOL` or `BACKVOL`. If `purgeVol` is set to a non-zero value, then that volume will be purged during the clone operation. This may be more efficient that separate clone and purge calls when making backup volumes. The `newVol` parameter sets the new clone's ID. It is illegal to pass a zero in `newVol`.

Error Codes

`VOLSERBADNAME` The volume name contained in `newName` was longer than 31 characters plus the trailing null.

`VOLSERBAD_ACCESS` The caller is not authorized to clone a volume.

`ENOENT` The `fromTrans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

`VBUSY` The given transaction was already in use; indicating that someone else is currently manipulating the specified clone.

`EROFS` The volume associated with the given `trans` is read-only (either `ROVOL` or `BACKVOL`).

`EXDEV` The volume associated with the `trans` transaction and the one specified by `purgeVol` must be on the same disk device, and they must be cloned from the same parent volume.

`EINVAL` The `purgeVol` must be read-only, i.e. either type `ROVOL` or `BACKVOL`.

5.7.9 **AFSVolReClone** — Re-clone a volume

```
int AFSVolReClone(IN struct rx_connection *z_conn,
                  IN long tid,
                  IN long cloneID)
```

Description

Recreate an existing clone, with identifier `cloneID`, from the volume associated with transaction `tid`.

Error Codes

- `VOLSERBAD_ACCESS` The caller is not authorized to clone a volume.
- `ENOENT` The `tid` transaction was not found.
- `VOLSERTRELE_ERROR` The `tid` transaction's reference count could not be dropped to the proper level.
- `VBUSY` The given transaction was already in use; indicating that someone else is currently manipulating the specified clone.
- `EROFS` The volume to be cloned must be read-write (of type `RWVOL`).
- `EXDEV` The volume to be cloned and the named clone itself must be on the same device. Also, `cloneID` must have been cloned from the volume associated with transaction `tid`.
- `EINVAL` The target clone must be a read-only volume (i.e., of type `ROVOL` or `BACKVOL`).

5.7.10 AFSVolSetForwarding — Set forwarding info for a moved volume

```
int AFSVolSetForwarding(IN struct rx_connection *z_conn,
                        IN long tid,
                        IN long newsite)
```

Description

Record the IP address specified within `newsite` as the location of the host which now hosts the volume associated with transaction `tid`, formerly resident on the current host. This is intended to gently guide *Cache Manager*s who have stale volume location cached to the volume's new site, ensuring the move is transparent to clients using that volume.

Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to create a forwarding address.

`ENOENT` The `trans` transaction was not found.

5.7.11 AFSVolTransCreate — Create transaction for a [volume, partition]

```
int AFSVolTransCreate(IN struct rx_connection *z_conn,
                      IN long volume,
                      IN long partition,
                      IN long flags,
                      OUT long *trans)
```

Description

Create a new *Volume Server* transaction associated with volume ID `volume` on partition `partition`. The type of volume transaction is specified by the `flags` parameter. The values in `flags` specify whether the volume should be treated as busy (`ITBusy`), offline (`ITOffline`), or in shared read-only mode (`ITReadOnly`). The identifier for the new transaction built by this function is returned in `trans`.

Creating a transaction serves as a signal to other agents that may be interested in accessing a volume that it is unavailable while the Volume Server is manipulating it. This prevents the corruption that could result from multiple simultaneous operations on a volume.

Error Codes

`EINVAL` Illegal value encountered in `flags`.

`VOLSERVOLBUSY` A transaction could not be created, thus the given [volume, partition] pair was busy.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level after creation.

5.7.12 AFSVolEndTrans — End a transaction

```
int AFSVolEndTrans(IN struct rx_connection *z_conn,
                   IN long trans,
                   OUT long *rcode)
```

Description

End the transaction identified by `trans`, returning its final error code into `rcode`. This makes the associated [volume, partition] pair eligible for further *Volume Server* operations.

Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to create a transaction.

`ENOENT` The `trans` transaction was not found.

5.7.13 AFSVolGetFlags — Get volume flags for a transaction

```
int AFSVolGetFlags(IN struct rx_connection *z_conn,
                   IN long trans,
                   OUT long *flags)
```

Description

Return the value of the `vflags` field of the `struct volser_trans` object describing the transaction identified as `trans`. The set of values placed in the `flags` parameter is described in Section 5.2.3.1. Briefly, they indicate whether the volume has been deleted (`VTDeleted`), out of service (`VTOutOfService`), or marked delete-on-salvage (`VTDeleteOnSalvage`).

Error Codes

`ENOENT` The `trans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

5.7.14 AFSVolSetFlags — Set volume flags for a transaction

```
int AFSVolSetFlags(IN struct rx_connection *z_conn,
                   IN long trans,
                   IN long flags)
```

Description

Set the value of the `vflags` field of the `struct volser_trans` object describing the transaction identified as `trans` to the contents of `flags`. The set of legal values for the `flags` parameter is described in Section 5.2.3.1. Briefly, they indicate whether the volume has been deleted (`VTDeleted`), out of service (`VTOutOfService`), or marked delete-on-salvage (`VTDeleteOnSalvage`).

Error Codes

`ENOENT` The `trans` transaction was not found.

`EROFS` Updates to this volume are not allowed.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

5.7.15 AFSVolGetName — Get the volume name associated with a transaction

```
int AFSVolGetName(IN struct rx_connection *z_conn,
                  IN long tid,
                  OUT char **tname)
```

Description

Given a transaction identifier `tid`, return the name of the volume associated with the given transaction. The `tname` parameter is set to point to the address of a string buffer of at most 256 chars containing the desired information, which is created for this purpose. **Note:** the caller is responsible for freeing the buffer pointed to by `tname` when its information is no longer needed.

Error Codes

`ENOENT` The `tid` transaction was not found, or a volume was not associated with it (`VSrv` internal error).

`E2BIG` The volume name was too big (greater than or equal to `SIZE` (1,024) characters.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

5.7.16 AFSVolGetStatus — Get status of a transaction/volume

```
int AFSVolGetStatus(IN struct rx_connection *z_conn,
                    IN long tid,
                    OUT struct volser_status *status)
```

Description

This routine fills the `status` structure passed as a parameter with status information for the volume identified by the transaction identified by `tid`, if it exists. Included in this status information are the volume's ID, its type, disk quotas, the IDs of its clones and backup volumes, and several other administrative details.

Error Codes

`ENOENT` The `tid` transaction was not found.

`VOLSERTRELE_ERROR` The `tid` transaction's reference count could not be dropped to the proper level.

5.7.17 AFSVolSetIdsTypes — Set header info for a volume

```
int AFSVolSetIdsTypes(IN struct rx_connection *z_conn,
                      IN long tId
                      IN char *name,
                      IN long type,
                      IN long pId,
                      IN long cloneId,
                      IN long backupId)
```

Description

The transaction identifed by `tId` is located, and the values supplied for the volume `name`, volume `type`, parent ID `pId`, clone ID `cloneId` and backup ID `backupId` are recorded into the given transaction.

Error Codes

`ENOENT` The `tId` transaction was not found.

`VOLSERBAD_ACCESS` The caller is not authorized to call this routine.

`VOLSERBADNAME` The volume name contained in `name` was longer than 31 characters plus the trailing null.

`VOLSERTRELE_ERROR` The `tId` transaction's reference count could not be dropped to the proper level.

5.7.18 AFSVolSetDate — Set creation date in a volume

```
int AFSVolSetDate(IN struct rx_connection *z_conn,
                  IN long tid,
                  IN long newDate)
```

Description

Set the `creationDate` of the `struct volintInfo` describing the volume associated with transaction `tid` to `newDate`.

Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to call this routine.

`ENOENT` The `tId` transaction was not found.

`VOLSERTRELE_ERROR` The `tid` transaction's reference count could not be dropped to the proper level.

5.7.19 AFSVolListPartitions — Return a list of AFS partitions on a server

```
int AFSVolListPartitions(IN struct rx_connection *z_conn,
                         OUT struct pIDs *partIDs)
```

Description

Return a list of AFS partitions in use by the server processing this call. The output parameter is the fixed-length `partIDs` array, with one slot for each of 26 possible partitions. By convention, AFS partitions are named `/vicepx`, where *x* is any letter. The `/vicepa` partition is represented by a zero in this array, `/vicepa` by a 1, and so on. Unused partitions are represented by slots filled with a -1.

Error Codes

`---` None.

5.7.20 AFSVolPartitionInfo — Get partition information

```
int AFSVolPartitionInfo(IN struct rx_connection *z_conn,
                        IN char *name,
                        OUT struct diskPartition *partition)
```

Description

Collect information regarding the partition with the given character string `name`, and place it into the `partition` object provided.

Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to call this routine.

`VOLSERILLEGAL_PARTITION` An illegal partition was specified by `name`

5.7.21 AFSVolListVolumes — Return a list of volumes on the server

```
int AFSVolListVolumes(IN struct rx_connection *z_conn,
                      IN long partID,
                      IN long flags,
                      OUT volEntries *resultEntries)
```

Description

Sweep through all the volumes on the partition identified by `partid`, filling in consecutive records in the `resultEntries` object. If the `flags` parameter is set to a non-zero value, then full status information is gathered. Otherwise, just the volume ID field is written for each record. The fields for a `volEntries` object like the one pointed to by `resultEntries` are described in Section 5.4.6, which covers the `struct volintInfo` definition.

Error Codes

`VOLSERILLEGAL_PARTITION` An illegal partition was specified by `partID`

`VOLSERNO_MEMORY` Not enough memory was available to hold all the required entries within `resultEntries`.

5.7.22 AFSVolListOneVolume — Return header info for a single volume

```
int AFSVolListOneVolume(IN struct rx_connection *z_conn,
                        IN long partID,
                        IN long volid,
                        OUT volEntries *resultEntries)
```

Description

Find the information for the volume living on partition `partID` whose ID is `volid`, and place a single `struct volintInfo` entry within the variable-size `resultEntries` object.

This is similar to the *AFSVolListVolumes()* call, which returns information on *all* volumes on the specified partition. The full volume information is always written into the returned entry (equivalent to setting the `flags` argument to *AFSVolListVolumes()* to a non-zero value).

Error Codes

`VOLSERILLEGAL_PARTITION` An illegal partition was specified by `partID`

`ENODEV` The given volume was not found on the given partition.

5.7.23 AFSVolGetNthVolume — Get volume header given its index

```
int AFSVolGetNthVolume(IN struct rx_connection *z_conn,
                       IN long index,
                       OUT long *volume,
                       OUT long *partition)
```

Description

Using `index` as a zero-based index into the set of volumes hosted by the server chosen by the `z_conn` argument, return the `volume` ID and `partition` of residence for the given index.

Note: This functionality has not yet been implemented.

Error Codes

`VOLSERNO_OP`	Not implemented.

5.7.24 AFSVolMonitor — Collect server transaction state

```
int AFSVolMonitor(IN struct rx_connection *z_conn,
                  OUT transDebugEntries *result)
```

Description

This call allows the transaction state of a *Volume Server* to be monitored for debugging purposes. Anyone wishing to supervise this *Volume Server* state may call this routine, causing all active transactions to be recorded in the given `result` object.

Error Codes

`---` None.

Bibliography

[1] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[2] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[3] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[4] Information Technology Center, Carnegie Mellon University. *Ubik - A Library For Managing Ubiquitous Data*, ITCID, Pittsburgh, PA, Month, 1988.

[5] Information Technology Center, Carnegie Mellon University. *Quorum Completion*, ITCID, Pittsburgh, PA, Month, 1988.

Index