Palo Alto Research Center

Weighted Voting for Replicated Data

By David K. Gifford

XEROX

Weighted Voting for Replicated Data

by David K. Gifford*

CSL-79-14 September, 1979

In a new algorithm for maintaining replicated data, every copy of a replicated file is assigned some number of votes. Every transaction collects a read quorum of *r* votes to read a file, and a write quorum of *w* votes to write a file, such that $r+w$ is greater than the total number of votes assigned to the file. This ensures that there is a non-null intersection between every read quorum and every write quorum. Version numbers make it possible to determine which copies are current. The reliability and performance characteristics of a replicated file can be controlled by appropriately choosing *r, w,* and the file's voting configuration. The algorithm guarantees serial consistency, admits temporary copies in a natural way by the introduction of copies with no votes, and has been implemented in the context of an application system called Violet.

CR Categories: 4.3, 4.35, 4.33, 3.81

Key Words and Phrases: weighted voting, replicated data, quorum, file system, file suite, representative, weak representative, transaction, locking, computer network

* David Gifford is a graduate student at Stanford University, and this work was supported in part by the Xerox Corporation and by the Fannie and John Hertz Foundation. This paper was presented at the Seventh Symposium on Operating System Principles on December 12, 1979 in Pacific Grove, California.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

1. Introduction

The requirements of distributed computer systems are stimulating interest in keeping copies of the same information at different nodes in a computer network. Replication of data allows information to be located close to its point of use, either by statically locating copies in high use areas, or by dynamically creating temporary copies as dictated by demand. Replication of data also increases the availability of data, by allowing many nodes to service requests for the same information in parallel, and by masking partial system failures. Thus, in some cases, the cost of maintaining copies is offset by the performance, communication cost, and reliability benefits that replicated data affords.

We present a new algorithm for the maintenance of replicated files. The algorithm can be briefly characterized by the following description:

- Every copy of a replicated file is assigned some number of votes.
- Every transaction collects a read quorum of *r* votes to read a file, and a write quorum of *w* votes to write a file, such that $r+w$ is greater than the total number of votes assigned to the file.
- This ensures that there is a non-null intersection between every read quorum and every write quorum. There is always a subset of the representatives of a file whose votes total to *w* that are current.
- Thus, any read quorum that is gathered is guaranteed to have a current copy.
- Version numbers make it possible to determine which copies are current.

The algorithm has a number of desirable properties:

- It continues to operate correctly with inaccessible copies.
- It consists of a small amount of extra machinery that runs on top of a transactional file system. Although "voting" occurs as will become evident later in the paper, no complicated message based coordination mechanisms are needed.
- It provides serial consistency. In other words, it appears to each transaction that it alone is running. The most current version of data is always provided to a user.
- By manipulating *r*, *w*, and the voting structure of a replicated file, a system administrator can alter the file's performance and reliability characteristics.
- All of the extra copies of a file that are created, including temporary copies on users' local disks, can be incorporated into our framework.

The remainder of the paper is organized as five sections. Section 2 describes related work, and how the algorithm differs from previous solutions. The algorithm's environment, interface, and basic structure are introduced in Section 3. Refinements are offered in Section 4, including the introduction of temporary copies and a new locking technique. The Violet System, which contains an implementation of this proposal, and some performance considerations are discussed in Section 5. The final section is a brief conclusion. The appendix demonstrates that our algorithm maintains serial consistency [1].

The ideas in this paper are illustrated in Mesa, a programming language developed at the Xerox

Palo Alto Research Center [8]. Mesa is well suited for this task because it contains integrated support for processes, monitors, and condition variables [6]. To simplify this presentation some nonessential details have been omitted from the Mesa examples.

2. Related Work

Previous algorithms for maintaining replicated data fall into two classes. Some insist that every object has a primary site which assumes responsibility for update arbitration. Distributed INGRES [10] is such a system. This technique is simple, but relatively inflexible. Others do not employ distinguished sites for objects, and are more complex than primary site algorithms. SDD-1 [9] keeps all copies of an object up to date by sending updates via a communication system that will buffer messages over machine crashes. Thomas' proposal [11] only requires that a majority of an object's copies be updated, and includes voting.

Although we share the notion of voting, it is difficult to directly compare our algorithm with Thomas' because the two provide different services. Notably:

- We guarantee serial consistency for queries (read-only transactions), while Thomas' algorithm does not.
- We do not insist that a majority of an object's copies be updated.
- Thomas' algorithm does not employ weighted voters, which limits its flexibility.
- Thomas' algorithm is more complex because it addresses consistency issues as well as replication issues. We have separated the two, resulting in an algorithm that is easier to reason about and to implement.
- Our structure allows for the inclusion of temporary copies.

3. The Basic Algorithm

3.1 Environment

The concepts necessary for the implementation of our algorithm are modeled below as a *stable file system.* In Section 3.3 we build our algorithm for replicated data assuming the existence of such a system.

Our exposition uses two kinds of objects, *files* and *containers.* Files are arrays of bytes, addressed by read and write operations as described below. Containers are storage repositories for files; they are intended to represent storage devices such as disk drives. These objects, and others introduced later in the paper, have unique names. No two objects will ever be assigned the same name, even if they are on different machines. We will not concern ourselves further with how programs acquire names, but will assume that the names of containers and files of interest are at hand.

A file is logically an array of bytes that can be created, deleted, read, and written.

```
File.Create: PROCEDURE [container: Container.ID]
    RETURNS [file: File.ID];

File.Delete: PROCEDURE [file: File.ID];

File.Read: PROCEDURE [file: File.ID, startByte, count: INTEGER,
    buffer: POINTER];

File.Write: PROCEDURE [file: File.ID, startByte, count: INTEGER,
    buffer: POINTER];
```

To keep the discussion simple, we assume that file system primitives operate on remote and local files alike. This can be accomplished by encoding a file's location or container in its unique identifier, or by maintaining location hints for remote files. These details will not be considered further.

Transactions are used to define the scope of concurrency control and failure recovery. A transaction is a group of related file operations bracketed by a begin transaction call and a commit transaction call.

```
Transaction.Begin: PROCEDURE;

Transaction.Commit: PROCEDURE;
```

A transaction hides concurrency by making it appear to its file operations that there is no other activity in the system, a property known as *serial consistency* [1]. A transaction hides undesirable events that can be recovered from, such as a detected disk read error, or a server crash. A transaction also guarantees that either all of its write operations are performed, or none of them are. Furthermore, once a transaction has committed, its effects must be resilient to hardware failures, such as a server crash. Every process has a single current transaction. Thus, for an application program to use two transactions it must create at least two processes. A forked process can join its parent's transaction by calling:

```
Transaction.JoinParentsTransaction: PROCEDURE;
```

A file may be unavailable if the server it resides on is down, or if there is a communication failure. If a read operation is directed to a file that is unavailable, the corresponding File.Read call will never return. Multiple processes are used by our algorithm to allow it to proceed in this case. Outstanding uncompleted reads, because they never occurred, do not affect the ability of a transaction to commit. The transaction system only guarantees serial consistency for reads that have actually completed when the transaction is committed. Likewise, if a write operation is directed to a file that is unavailable, the corresponding File.Write call will never return. However, a transaction that attempts to commit with unfinished writes will remain uncommitted until all of its writes complete.

It is possible that a user will want to abort a transaction in progress. A transaction abort, which can be initiated by a user as shown below, will discard all of a transaction's writes, and terminate the transaction.

```
Transaction.Abort: PROCEDURE;
```

It is also possible that the file system will spontaneously abort a transaction because of a server crash, communication failure, or lock conflict.

This concludes our model set of primitive objects and operations. The model abstracts a confederation of cooperating computers into a structure that has uniform naming and a distributed transactional file system. As we shall see in following sections, the abstractions introduced here make the replication algorithm straightforward to explain. Of course we believe that the model that we have described is realizable and practical; in fact, the ideas necessary for an implementation have received a great deal of attention. Gray [4] provides a nice discussion of two phase commit protocols, locking, and synchronization primitives. Lampson and Sturgis [5, 7] describe an implemented system that has all of the properties our model requires.

3.2 Interface

Our algorithm uses the facilities described in Section 3.1 to provide an abstraction called a *file suite.* This is a file that is realized by a collection of copies, which we call *representatives* because of the democratic way in which update decisions are reached. When a file suite is created, a description of its configuration must be supplied, which includes *r, w,* the number of representatives, the containers where they should be stored, and the number of votes each should be accorded.

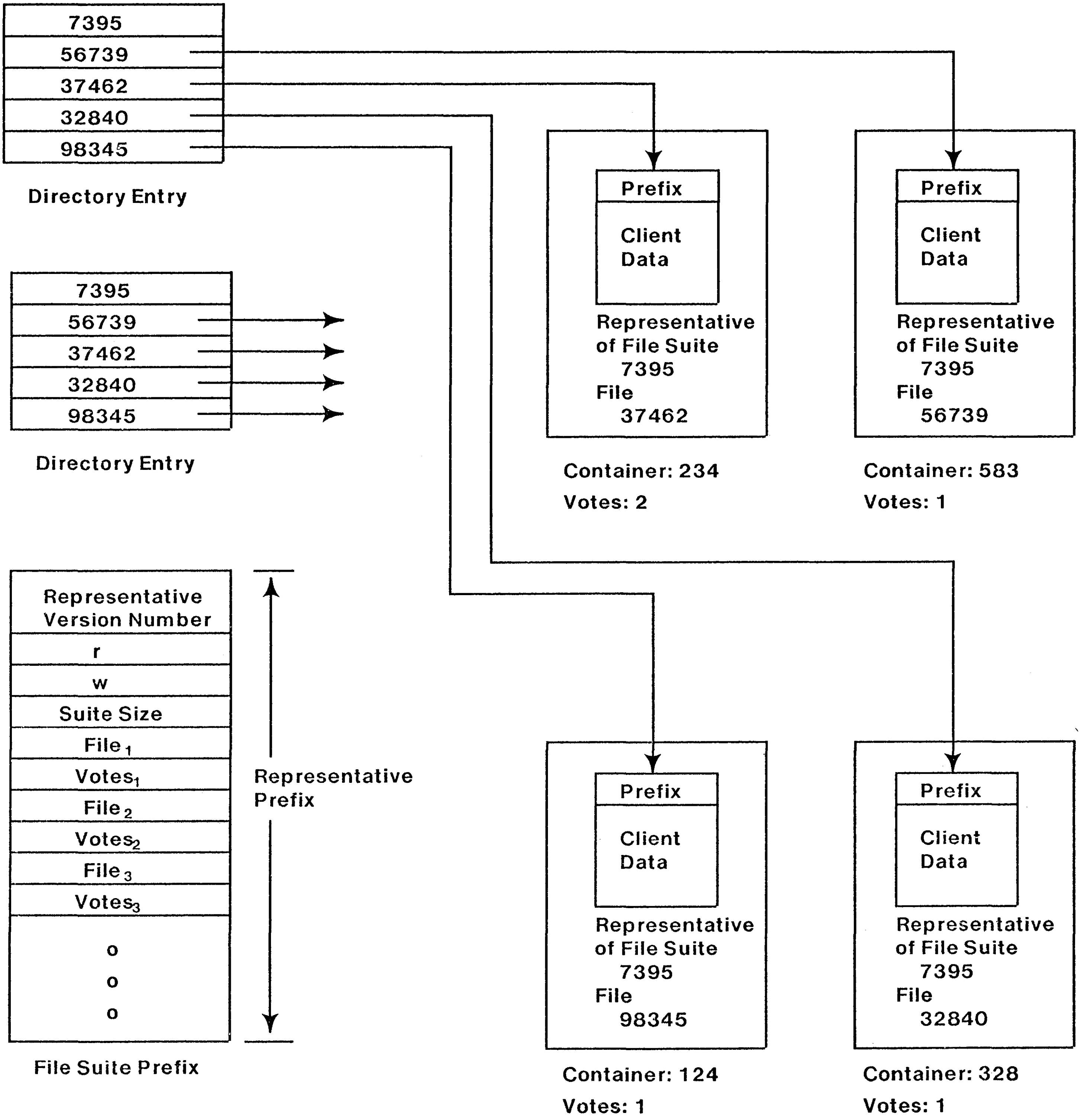
```
Configuration: TYPE = RECORD [
    r: INTEGER,
    w: INTEGER,
    v: ARRAY OF RECORD [container: Container.ID, votes: INTEGER]];

File.CreateSuite: PROCEDURE [configuration: Configuration]
    RETURNS [suite: File.ID];
```

File.CreateSuite stores a suite's configuration in stable storage. The structures stored would depend on the algorithm's implementation, but Figure 1 shows one possible alternative. A suite is cataloged by directory entries, preferably more than one in case one of them is unavailable. Each representative has a prefix that identifies all the other representatives in the suite and their voting strength.

Once created, a file suite can be treated like an ordinary file. The File.Read, File.Write, and File.Delete operations specified in Section 3.1 can be used to manipulate the abstract array of bytes represented by a file suite. Like file operations, all file suite operations are part of some transaction. A file suite appears to be an ordinary file in almost every respect.

Differences arise because a file suite can have performance and reliability characteristics that are impossible for a file. It is possible to tailor the reliability and performance of a file suite by manipulating its voting configuration. A high performance suite results by heavily weighting high performance representatives, and a very reliable suite results by heavily weighting reliable representatives. A file suite can also be made very reliable by having many equally weighted representatives. A completely decentralized structure results from equally weighting representatives, and a completely centralized scheme results from assigning of all of the votes to one representative. Thus the algorithm falls into both of the classes described in Section 2.



Example of a File Suite Configuration

Figure 1

Once the general reliability and performance of a suite is established by its voting configuration, the relative reliability and performance of Read and Write can be controlled by adjusting r and w. As w decreases, the reliability and performance of writes increases. As r decreases, the reliability and performance of reads increases. The choice of r and w will depend on an application's read to write ratio, the cost of reading and writing, and the desired reliability and performance.

The following examples suggest the diverse mix of properties that can be created by appropriately setting r and w. In the table below we assume that the probability that a representative is unavailable is .01.

Example 1 is configured for a file with a high read to write ratio in a single server, multiple user environment. Replication is used to enhance the performance of the system, not the reliability. There is one server on a local network that can be accessed in 75 milliseconds. Two users have chosen to make copies on their personal disks by creating weak representatives, or representatives with no votes (see Section 4.1 for a complete discussion of weak representatives). This allows them to access the copy on their local disk, resulting in lower latency and less traffic to the shared server.

Example 2 is configured for a file with a moderate read to write ratio that is primarily accessed from one local network. The server on the local network is assigned two votes, with the two servers on remote networks assigned one vote apiece. Reads can be satisfied from the local server, and writes must access the local server and one remote server. The system will continue to operate in read-only mode if the local server fails. Users could create additional weak representatives for lower read latency.

Example 3 is configured for a file with a very high read to write ratio, such as a system directory, in a three server environment. Users can read from any server, and the probability that the file will be unavailable is very small. Updates must be applied to all copies. Once again, users could create additional weak representatives on their local machines for lower read latency.

	Example 1	Example 2	Example 3
Latency (msec)			
Representative 1	75	75	75
Representative 2	65	100	750
Representative 3	65	750	750
Voting Configuration	⟨1, 0, 0⟩	⟨2, 1, 1⟩	⟨1, 1, 1⟩
r	1	2	1
w	1	3	3
Read			
Latency (msec)	65	75	75
Blocking Probability	$1.0 \text{ X } 10^{-2}$	$2.0 \text{ X } 10^{-4}$	$1.0 \text{ X } 10^{-6}$
Write			
Latency (msec)	75	100	750
Blocking Probability	$1.0 \text{ X } 10^{-2}$	$1.0 \text{ X } 10^{-2}$	$3.0 \text{ X } 10^{-2}$

3.3 The Algorithm

We present the basic algorithm in prose and fragments of Mesa code. The prose is meant be a complete explanation, with the Mesa code provided so the reader can check his understanding of the ideas. All the Mesa procedures shown below are part of a single monitor called FileSuite. There is a separate instance of FileSuite for each transaction accessing a given suite. ENTRY procedures manipulate shared data, and thus lock the monitor. Careful use of public non-entry procedures has been made so the monitor is never locked while input or output is in progress, allowing FileSuite to process simultaneous requests.

```
FileSuite: MONITOR [suiteName: File.ID] = BEGIN
    VersionNumber: TYPE = {unknown, 1, 2, 3, 4, ... }
    Set: TYPE = ARRAY OF BOOLEAN;
    SuiteEntry: TYPE = RECORD [
        name: File.ID,
        version: VersionNumber,
        votes: INTEGER];
    suite: ARRAY OF SuiteEntry;
    currentVersionNumber: VersionNumber;
    firstResponded: BOOLEAN;        -- true when first representative has responded
    r: INTEGER;     -- number of votes required for a read quorum
    w: INTEGER;     -- number of votes required for a write quorum
```

When FileSuite is instantiated, the number of representatives, their names, their version numbers, their voting strengths, *r*, and *w* must be copied from some representative's prefix into the data structure shown above. This information must be obtained with the same transaction that is later used to access the file suite, in order to guarantee that it accurately reflects the suite's configuration. Additional information, such as the speed of a representative, has been omitted from a SuiteEntry to make the basic algorithm easier to understand.

To read from a file suite, a read quorum must be gathered to ensure that a current representative is included. After a file suite is first accessed, collecting a file suite never encounters any delays. The operation of the collector which gathers a quorum is described in detail below. From the quorum, any current representative can actually be read. Ideally, one would like to read from the representative that will respond fastest.

```
Read: PROCEDURE [file: File.ID, firstByte, count: INTEGER, buffer: POINTER] =
    BEGIN
    -- select best representative
        quorum: Set ← CollectReadQuorum[];
        best: INTEGER ← SelectFastestCurrentRepresentative[quorum];
    -- send request and wait for response
        File.Read[suite[best].name, firstByte, count, buffer];
    END;
```

To write to a file suite, a write quorum is assembled: all of the representatives in the quorum must be current so updates are not applied to obsolete representatives. All of the writes to the quorum are done in parallel. The first write of a transaction increments the version numbers of its write quorum Thus, all subsequent writes will be directed to the same quorum because it will be the only one that is current. Determining which write is the first one must done under the protection of the monitor, and is not shown in the Mesa code. With the procedure below, the result of issuing two concurrent writes that update the same portion of a file is undefined.

```
Write: PROCEDURE [file: File.ID, firstByte, count: INTEGER, buffer: POINTER] =
    BEGIN
    -- select write quorum
        quorum: Set ← CollectWriteQuorum[];
        i, count: INTEGER ← 0;
        process: ARRAY OF PROCESS;
    -- send requests to all members of quorum, and wait for responses
        FOR i IN [1..LENGTH[suite]]
            DO
            IF quorum[i] THEN
                BEGIN
                count ← count + 1;
                process[count] ← FORK
                    RepresentativeWrite[i, firstByte, count, buffer];
                END;
            ENDLOOP;
        FOR i IN [1..count]
            DO
            JOIN process[i];
            ENDLOOP;
    END;
RepresentativeWrite: PROCEDURE [i, firstByte, count: INTEGER, buffer: POINTER] =
    BEGIN
    -- we are acting on behalf of our parent; join its transaction
        Transaction.JoinParentsTransaction[];
        UpdateVersionNumber[i];
    -- write data on representative and inform parent process
        File.Write[suite[i].name, firstByte, count, buffer];
    END;
```

It is possible that a representative will become unavailable while a file suite is in use, perhaps due to a server crash. A simple solution to this problem, not shown in the procedures above, is to abort the current transaction if Read or Write take more than a specified length of time. This will restart the suite, as described below.

Quorum sizes are the *minimum* number of votes that must be collected for read and write operations to proceed. It is possible to increase the performance of a file suite by artificially expanding a quorum with additional representatives. Once again, to reduce complexity, the procedures shown above do not use this approach.

When a file suite is first accessed, version number inquiries are sent to representatives. The information that results is used as the basis for future collector decisions. To determine the correct value of a file suite's current version number a read quorum must be established before the file suite can entertain requests. All representatives might not contain the current voting rules, but the algorithm will stabilize with the correct rules before a read quorum is established, as shown in Section 4.6. If a representative is unreachable its version number read will never return. This does not prohibit a user's transaction from committing, as described in Section 3.1.

```
InitiateInquiries: PROCEDURE =
    BEGIN
    i: INTEGER;
    -- find out the state of representatives
        FOR i IN [1..LENGTH[suite]]
            DO
            Detach[FORK Inquiry[i]];
            ENDLOOP;
    -- set currentVersionNumber and voting rules
        [ ] ← CollectRead[];
    END;

Inquiry: PROCEDURE [i: INTEGER] =
    BEGIN
    -- we are acting on behalf of our parent
        Transaction.JoinParentsTransaction[];
    -- find out the state of a representative
        NewRepresentative[ReadPrefixInformation[i]];
    END;

ReadPrefixInformation: PROCEDURE [i: INTEGER] RETURNS [i, version, rP, wP: INTEGER, v: ARRAY OF
        INTEGER] =
    BEGIN
        < read version number, r, w, and array of voting strenghts from the prefix of representative i >
    END;

NewRepresentative: ENTRY PROCEDURE [i, version, rP, wP: INTEGER, v: ARRAY OF INTEGER] =
    BEGIN
    j: INTEGER;
    -- update shared data and notify
        suite[i].versionNumber ← version;
    -- if this is new information, update suite
        IF version>currentVersionNumber THEN
            BEGIN
            currentVersionNumber ← version;
            r ← rP;  w ← wP;
            FOR j IN [1..LENGTH[suite]]
                DO
                suite[j].votes ← v[j];
                ENDLOOP;
            END;
        firstResponded ← TRUE;
        BROADCAST CrowdLarger;
    END;
```

The collector is used by every file suite operation to gather a quorum of representatives. Normally the collector selects what it considers to be the quorum that will respond the fastest, and returns immediately to its caller. Occasionally one of two problems will arise. First, it is possible that a read quorum of the suite's representatives have not reported their version numbers. In this case the collector can only wait for one of them to report in. The second potential problem is that a read quorum have reported their version numbers, but there is not a current write quorum. This can only occur if some representatives have not reported their version numbers. In this case if $r < w$ the collector will initiate a background process to copy the contents of the suite into one of the obsolete representatives that has reported in. It is always legal to copy the current contents of the file suite to an obsolete representative. Note that the copy process will be reading from the suite, in effect a recursive call, but there will be enough votes for this read-only operation to proceed. To minimize lock conflicts the background process should be run in a separate transaction. The background process signifies its completion by breaking the transaction of its parent.

```
CrowdLarger: CONDITION;            -- notifed when a new representative is available
CollectReadQuorum: ENTRY PROCEDURE RETURNS [quorum: Set] =
    BEGIN
    i, j, votes: INTEGER;
    index: ARRAY OF INTEGER;
    -- until the first representative responds we don't have a seed for the voting rules
        UNTIL firstResponded DO WAIT CrowdLarger ENDLOOP;
    -- an endless loop that only returns when a quorum has been established
        DO
    -- if we have a read quorum here, then the voting rules are current
        index ← SortRepresentativesBySpeed[];
        quorum ← ALL[FALSE];  votes ← 0;
    -- see if we can find a read quorum
        FOR i IN [1..LENGTH[suite]]
            DO
            j ← index[i];
            IF suite[j].versionNumber≠unknown THEN
                BEGIN
                quorum[j] ← TRUE;
                votes ← votes + suite[j].votes;
                IF votes>=r THEN RETURN[quorum];
                END;
            ENDLOOP;
    -- we can't find a quorum
        WAIT CrowdLarger;
        ENDLOOP;
    END;
CollectWriteQuorum: ENTRY PROCEDURE RETURNS [quorum: Set] =
    BEGIN
    i, j, votes, readVotes: INTEGER;
    index: ARRAY OF INTEGER;
    -- an endless loop that only returns when a quorum has been established
        DO
        index ← SortRepresentativesBySpeed[];
        quorum ← ALL[FALSE];  votes ← readVotes ← 0;
    -- see if we can find a write quorum
        FOR i IN [1..LENGTH[suite]]
            DO
            j ← index[i];
            IF suite[j].versionNumber≠unknown THEN
                BEGIN
                readVotes ← readVotes + suite[j].votes;
                IF suite[j].versionNumber=current THEN
                    BEGIN
                    quorum[j] ← TRUE;
                    votes ← votes + suite[j].votes;
                    IF votes>=w THEN RETURN[quorum];
                    END;
                END;
            ENDLOOP;
    -- we can't find a write quorum; if we have a read quorum update obsolete representatives
        IF readVotes>=r THEN
            BEGIN
            FOR i IN [1..LENGTH[suite]]
                DO
                IF suite[i].versionNumber≠unknown AND
                    suite[i].versionNumber≠currentVersionNumber THEN
                    BEGIN
                    suite[i].versionNumber ← unknown;
                    Copy.StartBackground[from: suiteName, to: suite[i].name];
                    END;
                ENDLOOP;
            END;
        WAIT CrowdLarger;
        ENDLOOP;
    END;
```

If a user decides to abort his transaction, or if the system spontaneously aborts a user's transaction the suite is no longer in a well defined state. The version number information it is holding is no longer guaranteed to be serially consistent with ensuing operations, and must be discarded and replaced by new inquiries. Asynchronous representative writes or version number reads still in progress must be aborted.

```
Initialize: PROCEDURE =
    -- called at transaction abort and suite start-up
    BEGIN
    -- reset the suite
        Reset[];
    -- stop outstanding requests
        AbortFileSuiteProcesses[];
        Transaction.Begin[];
        InitiateInquiries[];
    END;
```

```
Reset: ENTRY PROCEDURE =
    BEGIN
    -- invalidate state information
        FOR i IN [1..LENGTH[suite]]
                DO
                suite[i].versionNumber ← unknown;
                suite[i].votes ← 0;
                ENDLOOP;
        firstResponded ← FALSE;
        currentVersionNumber ← unknown;
    END;
```

At transaction commit the module instance is deleted.

4. Refinements

4.1 Weak Representatives

We can incorporate temporary copies into the algorithm by introducing representatives with no votes, called *weak representatives.* Such a representative will not change the quorums of a file suite, and thus can be introduced at any time. However, it can be included in any quorum and, when placed on a high speed storage device, can improve the performance of a file suite.

Because a weak representative has no votes, it bears no responsibility for the long term safekeeping of data. There will always be a write quorum of other representatives that contain current data. Thus, if an error is detected while accessing a weak representative an acceptable means of recovery is to invalidate it by setting its version number to be unknown. This property allows weak representatives to be stored outside of the stable file system. We have made the further simplification of insisting that weak representatives be unshared. To insure that users do not share weak representatives exclusive locks are used.

The simplified recovery and concurrency requirements of weak representatives allow us to store them in a less general file system than the one outlined in Section 3.1. In particular, they can be stored on a user's personal computer using a very simple mechanism. After a crash on a user's personal machine it is sufficient to invalidate all weak representatives that are locked.

4.2 Lock Compatibility

A disadvantage that our algorithm has in comparison with Thomas' is that locks are set by the stable file system to guarantee serial consistency, which reduces the amount of concurrency in the system. For example, a typical locking structure that is used to guarantee serial consistency has two types of locks, read and write. These locks are set on data items implicitly in response to file operations. The compatibility of locks is specified by the matrix:

	No Lock	Read	Write
No Lock	Yes	Yes	Yes
Read	Yes	Yes	No
Write	Yes	No	No

A transaction is suspended if it attempts to set a lock that is incompatible. This matrix corresponds to the familiar rule that either *n* readers or one writer are permitted to access a file simultaneously.

This locking rule potentially can introduce long periods of time when information is unavailable. For example, if a user controls the length of a transaction, he can hold a write lock for a long period of time. This may naturally occur as a user thinks at the keyboard. To insure that no user monopolizes a file, a transaction will be timed out if other users are waiting for a file it has locked. A transaction that times out leaves files unchanged, because it is aborted. The same mechanism insures that cyclic lock dependencies (deadlocks) will be resolved by aborting some transaction. Time-outs did not provide an adequate solution in our environment, as we describe in Section 5.1.

A property of serial consistency is that all of a transaction's writes appear to occur at transaction commit time. We can take advantage of this property to increase the concurrency in our system. Writes appear to occur when they are issued, but in fact are buffered until commit time by the stable file system. A read following a write will receive the write's data. When a user writes a datum, an I-Write lock is set, for intention to write. At commit time I-Write locks are converted to Commit locks, and the writing actually takes place. The new lock compatibility matrix is:

	No Lock	Read	I-Write	Commit
No Lock	Yes	Yes	Yes	Yes
Read	Yes	Yes	Yes	No
I-Write	Yes	Yes	No	No
Commit	Yes	No	No	No

With this revised locking matrix, data is only unavailable for predictably short periods of time, during commit processing. This results in increased concurrency, as we discuss in Section 5.1. However, it may cause the later abortion of a transaction.

We chose to make multiple I-Write locks incompatible, because eventually one of the two transactions would probably commit, and become incompatible. Thus we chose not to postpone the inevitable.

4.3. Lower Degrees of Consistency

We have assumed that the algorithm must be capable of providing serial consistency. Lower degrees of consistency are possible, and allow liberties to be taken, for example, setting *r* to be 0. This corresponds to the notion "give me the latest version you can find, but I don't care if it isn't fresh". Certain applications that have self-correcting characteristics, such as name lookup, can use lower degrees of consistency. However, the unexplicable behavior of lower degrees precludes their widespread use. Gray et al [3] have explored the properties of various degrees of consistency in detail. Their Degree 3 consistency corresponds to serial consistency. If our algorithm is run on top of a file system that ensures Degree 0 or Degree 1 consistency, the algorithm will guarantee the same consistency it sees, a fact we will not prove here.

4.4 Size of Replicated Objects

The size of an object that is replicated should be chosen to match the needs of an intended application. For example, a data base manager might choose to replicate relations, tuples, or indexes. Each replicated object requires a version number. Because our algorithm depends on a file's version number remaining unchanged throughout a transaction, the smallest unit that can be individually locked is a file.

4.5 Broken Read Locks

If the stable file system can break read locks to resolve conflicts instead of aborting an entire transaction, two positive effects result. First, fewer transactions are aborted. Second, it is not necessary for a version number to remain unchanged during a transaction. If it does change, the broken read lock mechanism informs the algorithm. The smallest lockable unit is no longer a file, but instead is the smallest lockable unit supported by the stable file system.

4.6 Reassigning Votes

As the focus of references to a file suite changes, it is possible to update the file suite's voting configuration for optimum performance. Unless the refinement proposed in Section 4.5 is adopted, updating a suite's voting configuration conflicts with any other use of the suite. Section 3.3's inquiry process protects itself against such changes by reading a suite's voting structure under the protection of a transaction.

To change *r*, *w*, or the voting structure of a file suite it is necessary to ensure that that there is a write quorum under the new rules that is current. The change can then be effected by updating the prefixes of a set of representatives that is the union of a current write quorum and a future (under the new voting rules) write quorum. We claim that regardless of the order representatives are examined, the most recent voting rules will be discovered before a read quorum is established. Imagine that a transaction incorrectly assumes that an obsolete generation of voting rules, G, is current. Then there is a set voting rules, G+1, that is one generation more recent. But when G+1 was established its rules were written into a write quorum (under G). Therefore, the transaction would have examined a representative that contained or once contained G+1. If this representative did not contain G+1, then it contained a later generation of the voting rules. The version number mechanism in NewRepresentative would have replaced G with one of these later generations. But we assumed that the transaction did not find a later generation. We have our contradiction.

4.7 Replicating Containers

Using the procedure outlined in Section 3.2, a user specifies a collection of containers to create a file suite. This flexibility can complicate the operation of the system. Imagine that User A creates a file suite on containers C and D, and User B creates a file suite on containers D and E. If container D fails it is possible that one or both of the users could continue to access their suites, and it is also possible that both of the users could not proceed, depending on the voting configuration. In short, a system operator does not know what a failure of container D implies. Replicated containers provide a solution to this problem.

```
Container.CreateReplicated: PROCEDURE [configuration:
    Configuration] RETURNS [replicatedContainer: Container.ID];
```

A *replicated container* appears to the user exactly like an unreplicated one. However, when passed a replicated container, File.Create creates a file suite instead of a single file. Thus, the user is unaware that replication is taking place. We call the containers that compose a replicated container *base containers.*

This approach has several benefits:

- It is easy to determine the implications of removing a base container from service. This allows the system to be operated effectively.
- Voting structures can be tailored to the characteristics of the system's configuration by knowledgeable system administrators.
- Backup and archiving of replicated containers makes sense.
- Replicated containers can be mounted and dismounted as a unit. This is because all of a replicated container's corresponding base containers are easily identified.

4.8 Releasing Read Locks

Every lock that a transaction holds necessitates communication at commit time to ensure that the lock is still in force. Section 5's InitiateInquires procedure sends inquires to all representatives in the suite, to determine their status. A read lock is thus obtained on every representative that is available. An enhancement to the algorithm would be to release the read locks on representatives that are not used as part of a quorum before committing. This would reduce the amount of communication at commit time significantly.

4.9 Updating Representatives in Background

In conjunction with replicated containers, it is possible to operate servers that update obsolete representatives by examining a replicated container and initiating appropriate transfers. This can be done when there is surplus communication capacity in the internetwork.

5. Implementation

5.1 The Violet System

We have implemented the algorithm in the context of a simple decentralized data management system called Violet [2]. Violet is a distributed calendar system which provides simple relational views of personal and public calendars. Figure 4 is a picture of Violet's display-based user interface. A user interacts with Violet by selecting items from the command menu at the bottom of the screen.

Violet is designed to operate in the environment shown in Figure 2. Each user has a personal computer, with a bit-map display, a pointing device, and a local network interface. Local network segments operate at 2.84 megabits per second, and are connected together by gateways to form an internetwork.

Violet's implementation of file suites closely parallels the code fragments of Section 3.3; it creates and manipulates uniformly weighted representatives. Each file suite is managed by a monitored module, consisting of seven pages of Mesa source code. Instead of employing the directory suggested by Figure 1, a file suite name in Violet is a list of the representatives that compose the suite.

Section 3.1's stable file system is implemented by DFS [5, 7]. DFS is a system composed of cooperating servers that provide a decentralized transactional file system. The interface to DFS closely parallels the model we presented in Section 3.1. At the File interface, we found that the read latency of a 512 byte page on a local DFS (on the same local network as the user) was 75 milliseconds. When the server was located on a remote network, accessed through a 9.6 KB data connection, the read latency of a 512 byte page was 650 milliseconds. By comparison, the read latency of the user's local disk was 65 milliseconds.

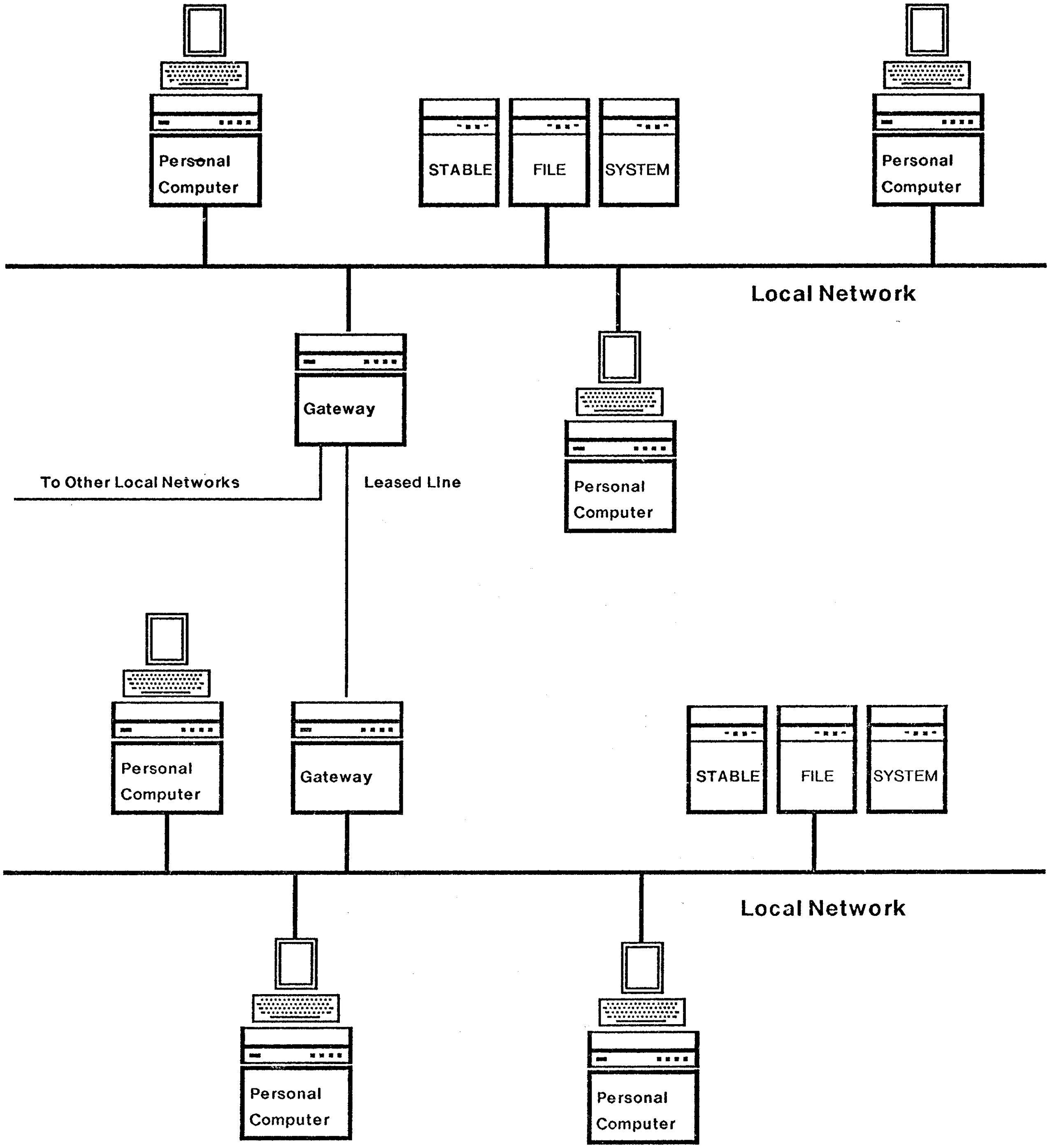
Replication is accomplished below Violet's simulated virtual memory, as shown in Figure 3. Pages from local and remote file suites are buffered in Level 2.

Before we implemented the proposal of Section 4.2, Violet exhibited the following undesirable behavior. Imagine that Users A and B are viewing the same calendar for a considerable length of time, longer than any minimum amount of time which the file system guarantees for viewing data. User A decides to update his view. As soon as User A writes into the view, User B's transaction will break. User B is now denied access to the data, and User B's machine constantly requests access to fresh information to repaint its screen. Meanwhile, User A holds a write lock while he thinks at the keyboard. User B eventually times out User A's lock, breaking A's transaction. Both screens repaint with the old information, and no net progress is made. This problem was solved by implementing Section 4.2's proposal.

5.2 Performance

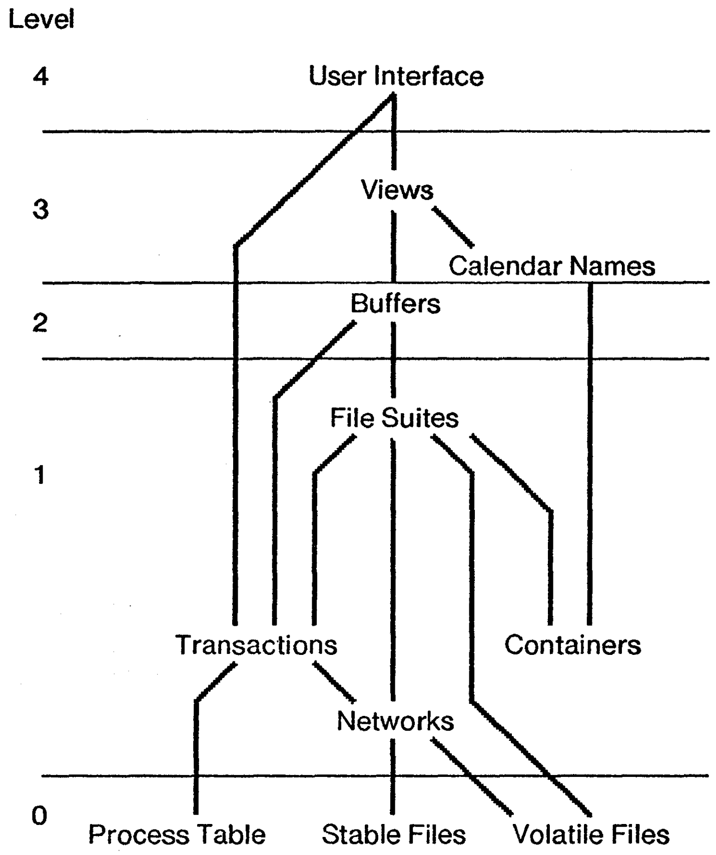
We present two sets of cost figures for our algorithm. The first set of costs, the number of reads and writes, are inherent in the algorithm. The second set, the number of messages and round-trip delay times, are the result of an implementation that uses DFS.

Consideration should be given to the source of the message and delay statistics. DFS was optimized for a local network, and thus some of the figures below are misleading. DFS requires



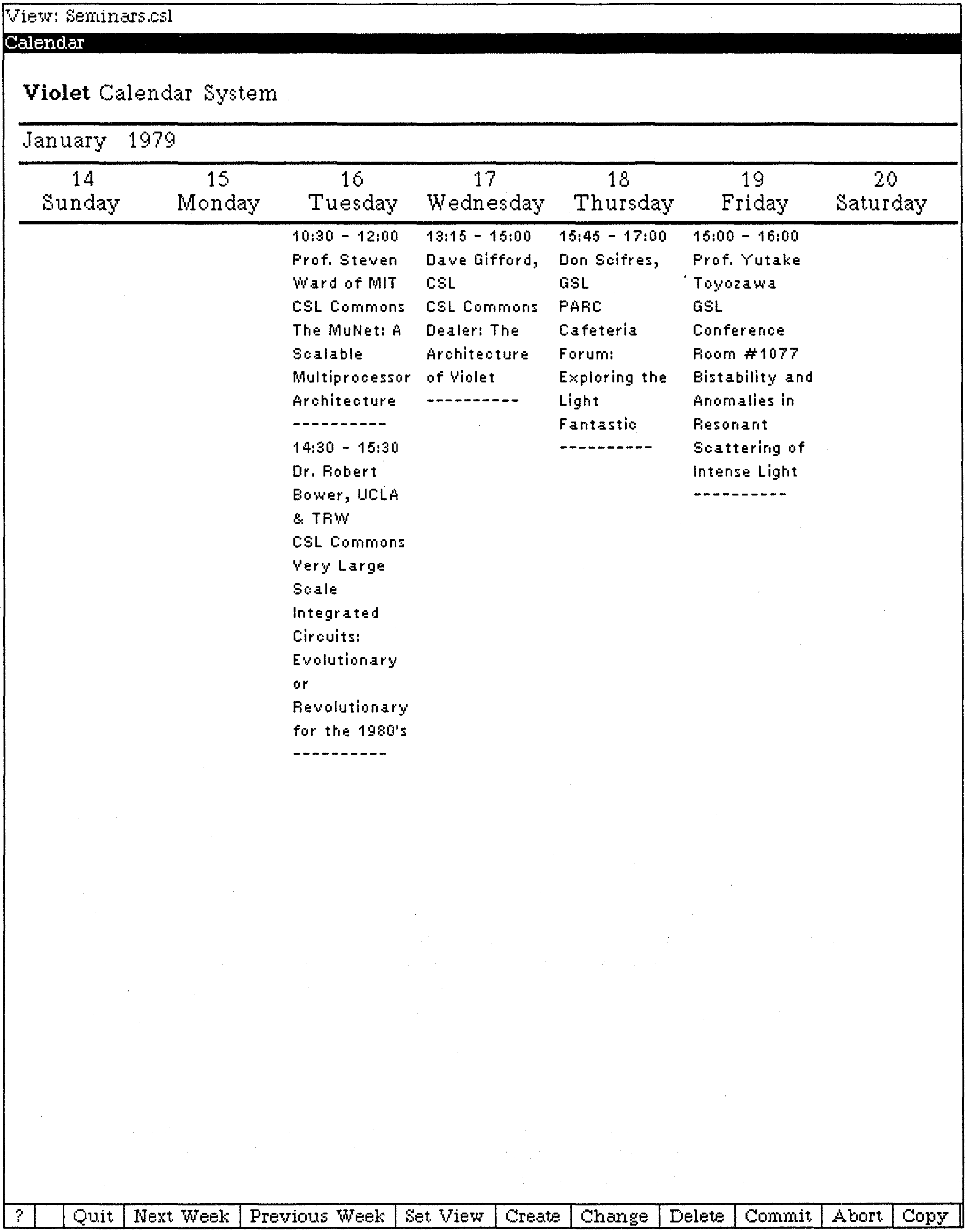
Typical Internetwork Environment

Figure 2



The Internal Structure of Violet

Figure 3



Violet Calendar System

January 1979

14 Sunday	15 Monday	16 Tuesday	17 Wednesday	18 Thursday	19 Friday	20 Saturday
		10:30 - 12:00 Prof. Steven Ward of MIT CSL Commons The MuNet: A Scalable Multiprocessor Architecture ---------- 14:30 - 15:30 Dr. Robert Bower, UCLA & TRW CSL Commons Very Large Scale Integrated Circuits: Evolutionary or Revolutionary for the 1980's ----------	13:15 - 15:00 Dave Gifford, CSL CSL Commons Dealer: The Architecture of Violet ----------	15:45 - 17:00 Don Scifres, GSL PARC Cafeteria Forum: Exploring the Light Fantastic ----------	15:00 - 16:00 Prof. Yutake Toyozawa GSL Conference Room #1077 Bistability and Anomalies in Resonant Scattering of Intense Light ----------	

Figure 4

three messages for a single request due to the protocol it uses for duplicate suppression. In addition, DFS uses a three phase commit protocol, while only a two phase protocol is logically necessary.

All of the figures shown are for worst case behavior. We have assumed that each representative is stored on a separate server, $r=w$, and there are at least two representatives. The Add Server figures refer to the cost of causing other servers to participate in an existing transaction. The Inquiries line represents the cost of the version number reads. Coordinator and Workers refer to the participants in commit processing.

d: a round trip delay time in the network
m: the number of representatives in a quorum

Command	Reads	Writes	Messages	Delay
Begin Transaction			3	d
First Read	$m+1$	0	$8m$-2	$4d$
Add Server	0	0	$5(m$-1)	$2d$
Inquiries	m	0	$3m$	d
Read	1	0	3	d
Subsequent Read	1	0	3	d
Subsequent Write	0	m	$3m$	d
End Transaction			$6m+3$	$3d$
Coordinator			3	d
Workers			$6m$	$2d$

6. Conclusion

We have demonstrated a new algorithm for the maintenance of replicated data that offers many benefits not provided by previous solutions. The introduction of weighted voting allows file suites to be synthesized with desired properties, including the presence of temporary copies. The separation of consistency considerations from replication has resulted in a conceptually simple approach which guarantees serial consistency in a straightforward way and is relatively easy to implement.

The facilities of Mesa have allowed us to express and implement a complex concurrent control structure. We invite language designers attempting to provide facilities for concurrent programming to gauge the difficulty of implementing the algorithm in their language.

The idea of weighting votes will undoubtedly have applications outside of replication algorithms. For example, when a decision has to be made by cooperating nodes with different probabilities of being correct, weighting the nodes' responses will improve the probability that a correct decision is reached.

Acknowledgments. I would like to thank my advisors, Butler Lampson and Susan Owicki, for their hard questions and encouragement during the course of this research. James Gray, Hector Garcia-Molina, Lawrence Stewart, Howard Sturgis, and the referees provided helpful comments and discussions.

References

1. Eswaran, K.P. et al The Notions of Consistency and Predicate Locks in a Database System, *Comm. ACM* 19, *11* (November 1976), pp. 624-633.

2. Gifford, D.K. Violet, An Experimental Decentralized System, Integrated Office System Workshop, IRIA, Rocquencourt, France, November, 1979. Available as CSL Report 79-12, Xerox Palo Alto Research Center, 1979.

3. Gray, J.N. et al Granularity of Locks and Degrees of Consistency in a Shared Data Base, in *Modeling in Data Base Management Systems*, North Holland Publishing, 1976, pp. 365-394.

4. Gray, J.N. Notes on Data Base Operating Systems, in *Operating Systems, An Advanced Course*, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, pp. 393-481.

5. Israel, J.E., Mitchell, J.G., and Sturgis, H.E. Separating Data From Function in a Distributed File System, Second International Symposium on Exploratory Systems, IRIA, Rocquencourt, France, October, 1978.

6. Lampson, B.W., and Redell, D.D. Experience with Processes and Monitors in Mesa, to appear in *Proceedings of the Seventh Symposium on Operating System Principles, ACM Operating Systems Review.*

7. Lampson, B.W., and Sturgis, H.E. Crash Recovery in a Distributed Data Storage System, *Comm. ACM,* to appear.

8. Mitchell, J.G. et al, *Mesa Language Manual,* CSL Report 79-3, Xerox Palo Alto Research Center, February, 1979

9. Rothnie, J.B., Goodman, N., and Bernstein, P.A., The Redundant Update Methodology of SDD-1: A System for Distributed Databases (The Fully Redundant Case), Rep. No. CCA-77-02, Computer Corporation of America, 1977.

10. Stonebraker, M. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES, *IEEE Trans. on Soft. Eng.* 5, *3* (May 1979), pp. 188-194

11. Thomas, R.H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases, *ACM Trans. on Database Systems* 4, *2* (June 1979), pp. 180-209.

Appendix: Consistency Considerations

Given the serial consistency [1] of stable files, which we allowed ourselves in Section 3.1, it is relatively straightforward to demonstrate that file suites also provide serial consistency. To do so, we need to formalize sufficient conditions for serial consistency, and show that file suites satisfy these conditions.

We formalize serial consistency as follows. A processing of an act is said to be *uninterrupted* if no other activities take place while it is in progress; a *concurrent* processing implies that other acts may be processed in parallel. An act is said to *appear to occur atomically* if the concurrent processing of the act has the same result that an uninterrupted processing would have produced. It follows that for a transaction mechanism to guarantee serial consistency a transaction must appear to occur atomically. A fresh read is one that, if issued now, would have the same result that it originally had. We propose the following axioms as sufficient conditions to guarantee serial consistency:

> TC1 (Atomicity of writes) All of the writes of a transaction appear to occur atomically at transaction commit.
>
> TC2 (Atomicity of reads) For a transaction to commit, at the instant the writes occur all of the transaction's reads must be fresh.

THEOREM. If a transaction obeys TC1 and TC2 then the transaction appears to occur atomically.

> PROOF. By contradiction. We assume that the operations comprising transaction *t* did not occur atomically. TC1 guarantees the atomicity of writes. Thus there is some read *r* that did not appear to occur at commit time. This implies that *r* would have produced different results at commit time. TC2 does not permit this. □

We demonstrate that our algorithm, coupled with the consistency of stable files, guarantees the consistency of file suites, by showing that a file suite obeys TC1 and TC2, and thus is consistent.

THEOREM. Our replication algorithm guarantees TC1 and TC2.

> PROOF. (TC1) We have assumed that all writes to stable files in a transaction appear to occur atomically. File suites writes are transformed to stable file writes. The desired result follows.
>
> (TC2) Imagine that our algorithm does not guarantee TC2. This implies that one of the file suite reads must not be fresh at commit time. This implies that some file suite read, if repeated now, would not yield the same results as it originally did. However, a key property of our algorithm is the guarantee that an updated datum will appear in a write quorum. But every read quorum and every write quorum have a non-null intersection. Hence at least one of our version number reads would not be fresh. But TC2 for the stable file system guarantees that all of the representative reads are fresh. We have our contradiction. □

XEROX

XEROX

Weighted Voting for Replicated Data

By David K. Gifford

CSL-79-14

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

XEROX® is a trademark of XEROX CORPORATION

Printed in U.S.A.