Weighted Voting for Replicated Data

David K. Gifford
Stanford University and Xerox Palo Alto Research Center

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.

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

CR Categories: 4.3, 4.35, 4.33, 3.81

The work reported here was supported in part by the Xerox Corporation, and by the Fannie and John Hertz Foundation.
Author's present address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1979 ACM 0-89791-009-5/79/1200/0150 $00.75

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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

- 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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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.

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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	1.0×10^{-2}	2.0×10^{-4}	1.0×10^{-6}
Write			
Latency (msec)	75	100	750
Blocking Probability	1.0×10^{-2}	1.0×10^{-2}	3.0×10^{-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 quorum 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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

```
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 strengths 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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.