46

An Algorithm for Data Replication

Timothy Mann, Andy Hisgen, Garret Swart

June 1, 1989

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

An Algorithm for Data Replication

Timothy Mann, Andy Hisgen, Garret Swart

June 1, 1989

Abstract

Replication is an important technique for increasing computer system availability. In this paper, we present an algorithm for replicating stored data on multiple server machines. The algorithm organizes the replicated servers in a master/slaves scheme, with one master election being performed at the beginning of each service period. The status of each replica is summarized by a set of monotonically increasing *epoch* variables. Examining the epoch variables of a majority of the replicas reveals which replicas have up-to-date data. The set of replicas can be changed dynamically. Replicas that have been off-line can be brought up to date in background, and *witness* replicas, which store the epoch variables but not the data, can participate in the majority voting. The algorithm does not require distributed atomic transactions. The algorithm also permits client machines to cache copies of data, with strict cache consistency being ensured by having the replicated servers keep track of which clients have cached what data. The work reported in this paper is part of an ongoing project to build a new replicated distributed file system with client caching, called *Echo*.

Contents

1 Introduction

As a distributed system grows, it tends to become less and less consistently available, because more and more different services running on different machines become vital to its function. At SRC, for example, the bootstrap service, the time service, the name service, and one or more file servers are all needed in the course of a typical user's working day, and the failure of any one of them can interrupt his progress. Nettled by this sort of problem, Leslie Lamport once defined a distributed system as a system in which the failure of a computer one has never heard of can make it impossible to get work done.

One way to combat this problem is to replicate the system's vital services—to construct each service from multiple interchangeable *replicas,* so that the service as a whole continues functioning even when some of the replicas are down. Today, SRC's bootstrap service, time service, and name service [3, 20] are replicated, and a small portion of the shared file system is also replicated in an *ad hoc* manner. The main difficulty in building such replicated services is keeping the replicas mutually consistent in spite of crashes, network partitions, malicious or bug-ridden clients, and the like. The difficulty increases with the amount of replicated state and the strictness of the consistency requirements; the examples above are listed in order of increasing difficulty. We have recently begun to work on a yet more difficult case of replication, a new file system (called *Echo*) in which all files are replicated, with guaranteed consistency even during updates.

This paper reports an early result of our work on Echo—the basic replication and caching algorithms. In our early design work, we wanted to abstract away the well-known problems that are common to all file systems, focusing instead on the problems of replication and caching. Therefore, we began by designing and implementing not a replicated, cached file system, but a replicated, cached array of integers. This effort has resulted in replication and caching algorithms that we have been able to extend and apply to the full file system.

In the remainder of this section we define the array interface, describe our system structure, and present an overview of the replication algorithm. Safety and progress conditions are given, and related work is surveyed. Section 2 develops the replication algorithm in detail, and Section 3 adds caching by client machines. Some extensions to the basic algorithms are presented in Section 4. Section 5 gives a summary and current status.

1.1 The `Ar` interface

Figure 1 gives the client interface to our replicated array of integers in Modula-2+ syntax [25]. We call the interface `Ar`—short for *Array.*

```
SAFE DEFINITION MODULE Ar;

TYPE
  Index = [1..100];

EXCEPTION
  ServiceUnavailable, LostUpdates;

PROCEDURE Get(i: Index): INTEGER
  RAISES {ServiceUnavailable};

PROCEDURE Set(i: Index; value: INTEGER)
  RAISES {ServiceUnavailable};

PROCEDURE Sync()
  RAISES {ServiceUnavailable, LostUpdates};

END Ar.
```

Figure 1: The `Ar` interface.

Informally, the semantics of this interface are as follows. The `Ar` service implements an array of integer variables with index ranging from 1 to 100. The call `Ar.Get(i)` returns the current value of the array element `i`, while the call `Ar.Set(i, value)` sets the value of element `i` to `value`. To the clients, there appears to be a single copy of the array, even though its implementation is replicated and the clients themselves run on many different machines. In other words, we obey *one-copy serializability*, meaning that the effect of a set of updates and queries on the replicated data is the same as the effect of applying the same set of updates and queries to a single copy of the data in some total order consistent with the partial orders seen by the clients [1, pp. 265–266].

The `Ar.Sync` procedure is present in this interface because we wanted to study the implementation and semantics of *write-behind* in our work on `Ar`. If write-behind is allowed, a call to `Ar.Set(i, value)` enqueues a new value to be written to the `i`th array element and immediately arranges that subsequent calls to `Ar.Get(i)` will return the new value, but it does not guarantee that the new value is stable. An inopportune crash may cause the new value to be forgotten, in which case the modified element reverts to its previous value. Whenever a client's call to `Ar.Sync` returns without raising an exception, `Ar` guarantees that the values written in all the client's calls to `Ar.Set` since the last previous `Ar.Sync` are stable, but if `Ar.Sync` raises the exception `LostUpdates`, one or more of the values written may have been lost. For the remainder of this section and in our discussion of replication (Section 2), we ignore the `Ar.Sync` procedure and assume that `Ar.Set` writes new values stably. We discuss write-behind together with caching, in Section 3.

Any of the routines in `Ar` can raise the exception `ServiceUnavailable`, which means that, despite its replication, the `Ar` service is at present unavailable to the caller.

Besides `Ar`, the service also exports an interface called `ArConfig`, which contains routines for reconfiguring the `Ar` implementation by adding or deleting replicas. These routines are discussed in Section 2.5.

1.2 System structure

Figure 2 shows the general structure of our replicated `Ar` implementation and gives the names of the components.

Shown at the bottom of the figure are several *server machines.* Each server machine has some private *stable storage,* writable storage that is highly likely to retain its contents across server crashes and restarts—a disk, for instance. The server can alter any set of bytes in stable storage in a single atomic operation—either all or none of the write is completed, even if the server crashes while the write is in progress. We assume that it is impossible for an unauthorized machine to impersonate a server.[1] Each server machine runs a software module called a *replica,* which manages a copy of the replicated array on the stable storage device.

Shown at the top of the figure are several *client machines.* Each can run *client programs* that use the `Ar` service. All the client programs on a machine call a *clerk* module on that machine that handles the details of

[1]The authentication mechanism needed to accomplish this is beyond the scope of this paper.

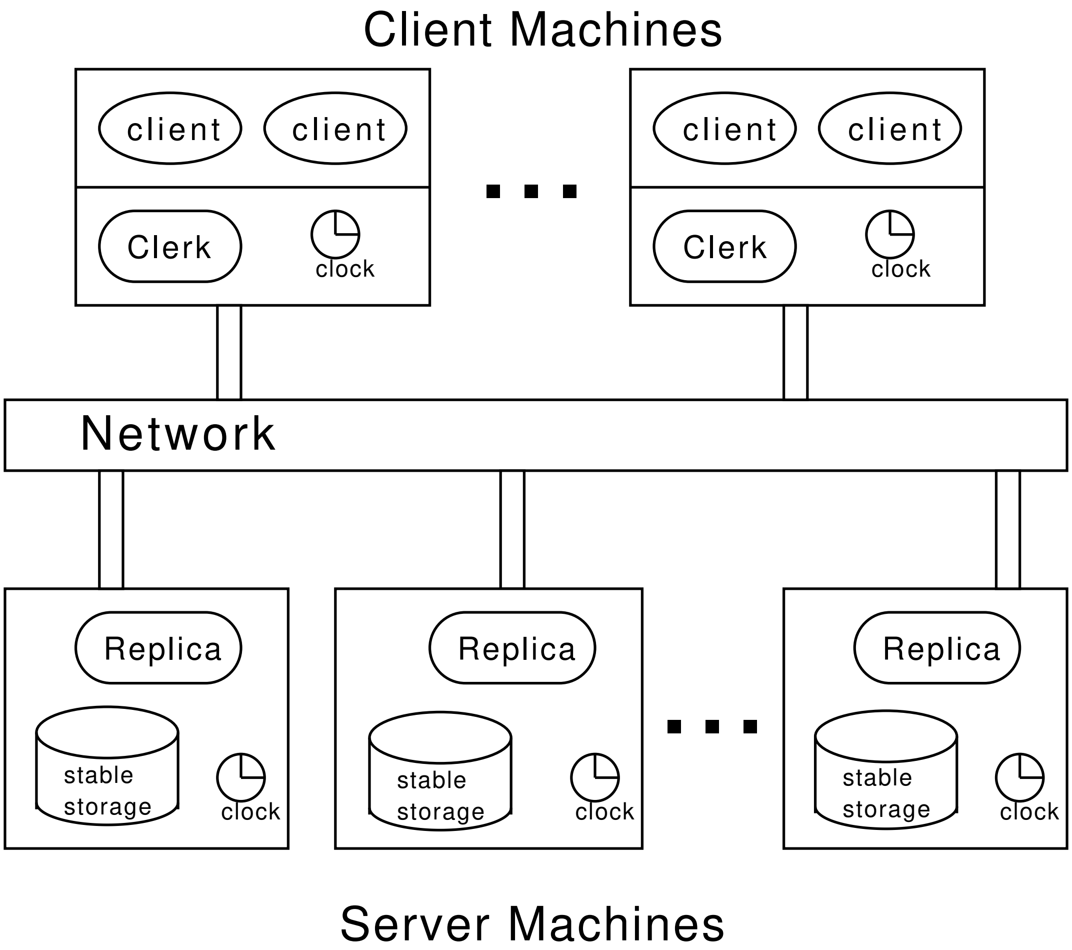

Figure 2: System components

communicating with the replicas. The clerk is also responsible for managing the client machine's cache.

The client and server machines are fully interconnected by a *network.*

Both client and server machines have *clocks* that can be used to measure time intervals or trigger events.

1.3 Algorithm overview

Our replication algorithm is based on *master election.* At any moment, the system either has exactly one master replica, or is trying to elect one. To be elected, a potential master must persuade a majority of the replicas (counting itself) to be its *slaves.* Normally, the slaves are exactly those replicas that are up and in communication with the master; whenever a replica crashes or an unenslaved replica comes up, a new election is held to reestablish this condition. Each election is followed by a recovery phase, in which the master ensures that its slaves agree with it on the current value of the replicated data. Once recovery is complete, the system resumes providing service to its clients, until the next election is triggered.

The master coordinates all operations. Each `Ar.Set` request passes from the calling client, through its clerk, to the master, and from there to all slaves. Each `Ar.Get` request passes from the client and clerk to the master, which satisfies it directly. (The caching and write-behind mechanism described in Section 3 allows most of a client's requests to be satisfied directly by its clerk, but cache misses and write-backs still follow this protocol. Section 4.1 extends the protocol to allow clerks to read from slaves.)

1.4 Safety and progress conditions

In this section we specify the fault tolerance of our algorithm by giving a set of conditions that the system components are required to meet; all "faults" that do not violate these conditions are tolerated. The conditions fall into two groups: those required in order for our implementation to behave correctly, called *safety conditions,* and those required for it to accomplish useful work, called *progress conditions.* If the safety conditions are violated, the implementation is free to do anything. Whenever the progress conditions are violated, the implementation is free to provide no service, but is not free to give wrong answers.

More precisely, we think of a fault-tolerant algorithm as a black box whose inputs are client requests and hardware component status (up, down, or faulty), and whose outputs are responses to client requests. The algorithm has a specification that determines which possible responses are correct; for `Ar`, we have given the specification informally in Section 1.1 above. The safety conditions define a predicate S on the system's hardware status (considered as a function of time), with the property that if S is true, client requests can never return incorrect responses; they either return correct responses or block forever. The algorithm's specification also identifies some correct responses as normal and some as exceptional (*i.e.,* undesirable); for `Ar`, responding without raising an exception is normal, while responding with an exception is exceptional. The progress conditions define a time-dependent predicate P on the system's hardware status, with the property that for any client request r submitted from client machine m at time t, there exists a time t' such that if P is true throughout the time interval $[t, t']$, then the system gives a normal response to the request at t'. There need be no way to determine t' in advance; this definition merely says that the request will complete normally in finite time if the progress predicate remains true continuously until the request completes.

Safety conditions. The safety predicate S for our `Ar` algorithm is the conjunction of the following conditions S1–S4, which must hold on every client and server machine at all times.

S1. The machines are fail-stop. That is, at any moment, each machine is either *up* or *down.* While a machine is up, it executes the correct algorithm—it does not exhibit Byzantine faults.[2] (However, we do not assume anything about the speed at which a machine executes its algorithm; in fact, if the algorithm is multithreaded, we allow each thread to be executed at arbitrary speed.) When a machine goes down (crashes), it ceases to update its stable storage or send messages to the network. When a machine comes up (restarts), it sets its volatile state to some predetermined, fixed value and resumes executing its algorithm.

S2. While a machine is up, its clock runs at approximately the rate of real time. That is, there is a constant $\rho \ll 1$, such that if $C_p(t_0)$ and $C_p(t_1)$ are the readings of machine p's clock at times t_0 and t_1 respectively, and p is up for the entire interval $[t_0, t_1]$, then

$$1 - \rho < \frac{C_p(t_1) - C_p(t_0)}{t_1 - t_0} < 1 + \rho$$

[2] A *Byzantine fault* is an arbitrary, perhaps even malicious failure of a system component; the term was coined by Lamport [18].

.

S3. A stable storage read either returns the results of the most recent write to the same stable storage cell, or raises an exception. A stable storage write either succeeds or raises an exception.[3]

S4. The network may duplicate, arbitrarily delay, or fail to deliver any message, but it does not alter or spontaneously generate messages.

Progress conditions. The progress predicate P for our `Ar` algorithm is the conjunction of the following conditions P1–P8, together with the safety predicate S. Recall that P is time-dependent—the system makes progress on an operation r submitted by machine m at time t when all these conditions are true, but may fail to make progress when any are false. We have cut a corner here by mentioning the system's internal state in P4; in principle we should replace this condition with one that mentions only previous hardware states and implies P4, but doing so would be cumbersome.

P1. Network connectivity is *static* and *transitive.* To define these terms, we first say that machines p and q are *connected* if all messages sent by p to q or by q to p are delivered within t_{net} seconds. Machines p and q are *disconnected* if no messages sent by p to q or by q to p are ever delivered. Network connectivity is then *static* if for all pairs of machines (p, q), either p and q are connected or they are disconnected. Connectivity is *transitive* if whenever p and q are connected, and q and r are connected, then p and r are also connected.

There is a set U of client and server machines such that:

P2. U contains a majority of the server machines.

P3. Stable storage operations on the server machines in U raise no exceptions.

P4. U contains at least one server with an up-to-date version of the array in its stable storage.

P5. The client machine m is in U.

P6. All members of U can complete any computation needed for the `Ar` algorithm within some fixed time t_{proc}.

P7. All pairs of machines in U are connected (as defined in P1).

P8. No message from a machine not in U arrives at a server machine in U.

---

[3] In our replication algorithm, if a read or write to a replica's stable storage raises an exception, the replica halts and never restarts.

In an actual installation, the number of server and client machines, the speed of the machines, and the speed of the network determine the values of t_{net} and t_{proc}. The reader should think of these values as being on the order of a second. Also, at most on the order of ten seconds of progress should be required to complete any operation requested.

These conditions are of course stricter than necessary. In P8, for example, we need only require that members of U receive no "confusing" messages, such as a message from a server not in U that triggers an election. But this condition is more difficult to formalize than the one given.

Byzantine client machines. Besides the form of safety we have just defined, our algorithm has another important safety-related property. Although condition S1 requires that all machines in the system be fail-stop at all times for the system to be safe at any time, our algorithm can in fact recover from transient Byzantine faults on client machines. The structure of the `Ar` algorithm makes it possible to model a Byzantine client machine as a non-faulty machine running a pseudo-client program that injects arbitrarily chosen `Ar.Set` requests into the system. Such pseudo-clients cause the real clients to see results that do not match the specification—but if the pseudo-clients are counted in with the real clients, the system's behavior is once again consistent with the specification. This property is useful because, at any future time when there are no client machines experiencing Byzantine faults, there are no pseudo-clients, and the real clients again see behavior that is consistent with the specification (though the values stored in the `Ar` array can have been changed arbitrarily during the faulty period).

The Echo file system has a similar but stronger property. Again, if a client machine is Byzantine, the only damage it can do to the system is to effectively inject requests from pseudo-client programs. But Echo makes use of an authentication service to guarantee that the servers will reject any such request unless it is made on behalf of a user who has actually logged into the Byzantine machine, and unless it requests an action that that user is authorized to perform.

Comparison with Byzantine fault model. Let us compare our fault model with the model used in the Byzantine agreement problem [9, 18].

Our fault model treats problems with the network explicitly, and our algorithm is safe provided that the network does not alter or spontaneously generate messages (condition S4). In the Byzantine approach, on the other hand, the network is modeled as being fault-free. Thus, when a Byzantine algorithm is implemented on a physical network that is subject to faults, a network fault must be modeled as a (Byzantine) fault in the machine that sent (or was to receive) the affected message. But because a Byzantine machine is free to do anything, Byzantine algorithms give no guarantees about the results that faulty machines achieve.

Our fault model is more restrictive than the Byzantine model—we permit only fail-stop crashes and timing faults.[4] We gain a great deal by making this restriction. Our algorithm guarantees safety in spite of any number of network or server faults including partition, yet it requires only one round of messages per `Ar.Set` during normal operation. A server that comes up again after a crash can be assumed not to have corrupted its stable storage (as it could have, had its fault been Byzantine), so it can quickly catch up to the current state rather than having to start from scratch.

Moreover, although Byzantine faults do happen occasionally in real systems, we believe there is little to be gained from attempting to tolerate those that do occur, at least in the replicated servers of a system like ours. We expect our servers to be physically secure, to run on reasonably well debugged hardware, and to run identical software on each replica. Physical security makes malicious server faults unlikely. With hardware that has advanced beyond the experimental stage, Byzantine hardware faults are also unlikely. (In our experience, most hardware bugs manifest themselves as crashes in any case.) This leaves software bugs as the most likely cause of Byzantine faults. But seeing that all the servers are running the same software, any bug present in one is present in all, so there is little hope of tolerating it through replication.

We have, however, made the effort to tolerate Byzantine faults in client machines (in the weak sense described above), because our assumptions about client machines are different from our assumptions about servers. We do not expect client machines to be physically secure or under centralized control; their users are free to run modified, perhaps even malicious versions of the clerk software. This assumption has put some constraints on our design—for example, it dictated the data flow on `Ar.Set` operations. We could not let the clerk broadcast operations directly to all replicas, because a Byzantine clerk might send different messages to different replicas, causing

[4] A *timing fault* is a failure to perform an action within the time that it was supposed to be performed, for example, taking longer than t_{net} seconds to deliver a message (progress condition P1 above).

their copies of the data to diverge. This problem could be avoided by using a Byzantine broadcast protocol, but we chose instead to save on messages by making use of our assumption that servers do not incur Byzantine faults.

1.5 Related work

We begin by reviewing three alternative approaches to implementing replication: multi-site atomic transactions, reliable atomic broadcast, and Lamport's state-machine approach. We then examine the tradeoff between consistency among replicas and availability. We close by discussing related work in election and caching.

Much of the existing work on replication employs multi-site atomic transactions as a building block: the existence of a multi-site atomic transaction facility is assumed, and replication is then built on top of it. This work includes that of Gifford [10] and most of the work surveyed in Chapter 8 of Bernstein et al. [1]. Our algorithm does not use multi-site transactions, because we were unwilling to pay the cost of running a multi-site atomic commit protocol for each replicated update. The most popular such protocol, two-phase commit, requires two rounds of messages per commit and blocks if the coordinator crashes or becomes unavailable at the wrong moment [1, pp. 226–236]. Another well-known protocol, a version of three-phase commit [1, pp. 256–259], is similar to our algorithm in its fault tolerance and blocking properties—roughly speaking, it is nonblocking as long as a majority of sites are up and communicating—but it requires three rounds of messages per commit, even in the absence of faults. In contrast, our algorithm requires only one round of messages per commit in the absence of faults. This is possible because our algorithm solves an easier problem: in our application, a replica never chooses to vote "no" on a proposed update, so aborts are caused only by faults.

Some replication work has taken the approach of building reliable atomic broadcast mechanisms and then layering replication on top of them [2, 5, 6, 21]. We are not attracted by this approach, because the broadcast mechanisms require several rounds of messages and/or a time delay before completion, and because this approach pushes the membership problem (reaching agreement on the set of sites that are up and should receive broadcasts) down into a lower layer, where its solution requires another elaborate mechanism.

Lamport [15] has proposed a general *state-machine approach* to replicating state in a distributed system. Each participating process runs its own copy of an application-specific state machine, while a general distributed algorithm provides the same sequence of events to each process so that consistent state transitions are triggered on each. Lamport has described two different fault-tolerant algorithms for communicating events in this approach—one based on a solution to the Byzantine generals problem [16], the second related to three-phase commit [17] and inspired in part by the `Ar` problem. The former algorithm is closely related to the atomic broadcast methods just discussed and similar comments apply. The latter algorithm, in its basic version, does not meet our requirements for `Ar`—both reads and writes require communicating with all replicas—but Lamport's paper describes some elaborations to the algorithm that seem to make it usable as an alternative to ours.

Our replication semantics are strict, in that we obey one-copy serializability. When replicas may be partitioned by communication faults, adhering to one-copy serializability effectively forbids providing service in more than one partition; otherwise, inconsistent updates could be performed in the different partitions. Like Thomas [26], we enforce this prohibition by requiring that a majority of replicas be up and in communication in order to provide service. Gifford generalized this majority consensus technique to quorum consensus, which permits trading off the number of replicas required for different operations; however, some operations still require more than one replica, so full service cannot always be provided [10]. It seems feasible to generalize our algorithm to use quorums as well, but we have not felt the need to do so.

Some work on replication has deliberately chosen looser replication semantics, accepting forms of consistency between replicas that are weaker than one-copy serializability, in order to achieve higher availability [7, 20, 24, 27]. Service may be provided with only one replica, and conflicting updates in different partitions are possible. Whether this tradeoff between data consistency and availability is acceptable is application-dependent. We prefer our stronger consistency semantics for a file system, because we see no way to resolve conflicting updates after partitions are reconnected without massive human or application assistance. For a name service [20], on the other hand, we believe that looser consistency semantics are acceptable—because the update rate is relatively low, most updates are made by system administrators, and much of the information may be structured as hints [19], which are explicitly permitted to be wrong. A survey of approaches to the problem of providing service while partitioned and coping with inconsistencies may be found in Davidson et al. [7].

Our algorithm uses an election to choose a master replica that has control of a majority. Elections have been widely studied [8]. Our election algorithm permits replicas to be added to or removed from the replica set; these operations are integrated with the election algorithm because they affect what it means to have a majority. (In most other work, changes to the replica set are made using multi-site atomic transactions.) Some researchers have explored a technique called dynamic voting, in which replicas that are down or not in communication with the majority are automatically removed from the replica set and can be automatically reinserted once they resume communication [13]. We have not adopted dynamic voting because the benefits it gives in return for its extra complexity are not compelling.

Our work incorporates a distributed caching algorithm, in which servers keep track of which client machines have cached what data and call them back when their caches must be invalidated. The algorithm is similar in many respects to those of the Andrew [12, 14] and Sprite [22] systems, and to one advanced by Burrows [4].

2 Replication Algorithm

We describe the replication algorithm in detail in this section. We begin by outlining the phases that the algorithm goes through in its operation and the states that each replica can be in, then go on to discuss each phase in turn. Initially, we give a simplified version of the algorithm that assumes the set of replicas is static and known to all replicas and clerks, but in Section 2.5, we explain how the full algorithm permits the set to be safely changed during operation. In Section 2.6 we describe how clerks find the current master and what they do when masterhood changes.

2.1 Phases and states

At any moment, a system running the `Ar` replication algorithm is in one of three global phases—**Service**, **Election**, or **Recovery**—as shown in Figure 3.

The system is normally in the **Service** phase, accepting and processing requests from clients.

The system moves to the **Election** phase whenever service is interrupted by the failure or recovery of a replica or the network. During this phase, the replicas hold an election to assemble a majority and choose one of their number as master.

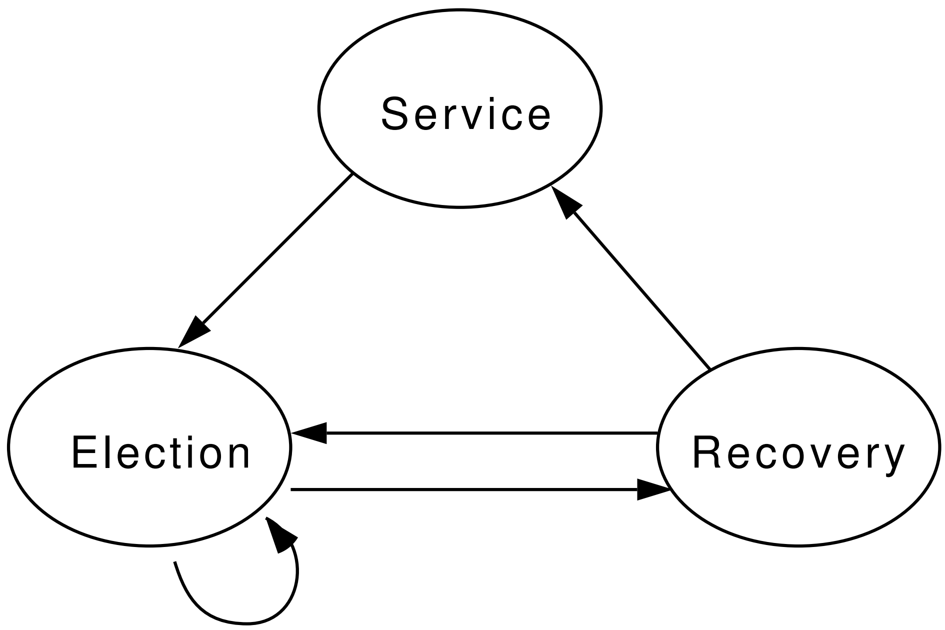

Figure 3: Phases in the replication algorithm

When an election succeeds, the replicas that participated in it are called *active* replicas, and the algorithm moves on to the **Recovery** phase. During recovery, the master directs the system in two interwoven tasks. First, the active replicas reconcile any differences among their copies of the data. Second, each active replica advances a set of *epoch* variables it holds in stable storage, so that inactive replicas can be identified as out-of-date during the next election. When these tasks have succeeded, the system reenters the **Service** phase.

Because the `Ar` replication algorithm is distributed and fault-tolerant, there is no single place where one can look to determine what phase it is in at any given moment. Rather, the phase at any given moment is a function of the states of all the replicas. There are five states a replica can be in, illustrated in Figure 4 and listed below.

- **Free**. A replica in the **Free** state is not a slave to any replica. A free replica continuously runs the `Ar` election algorithm, and moves into the **PotentialMaster** state if the algorithm computes that it is the best candidate for master. While running the election algorithm, a

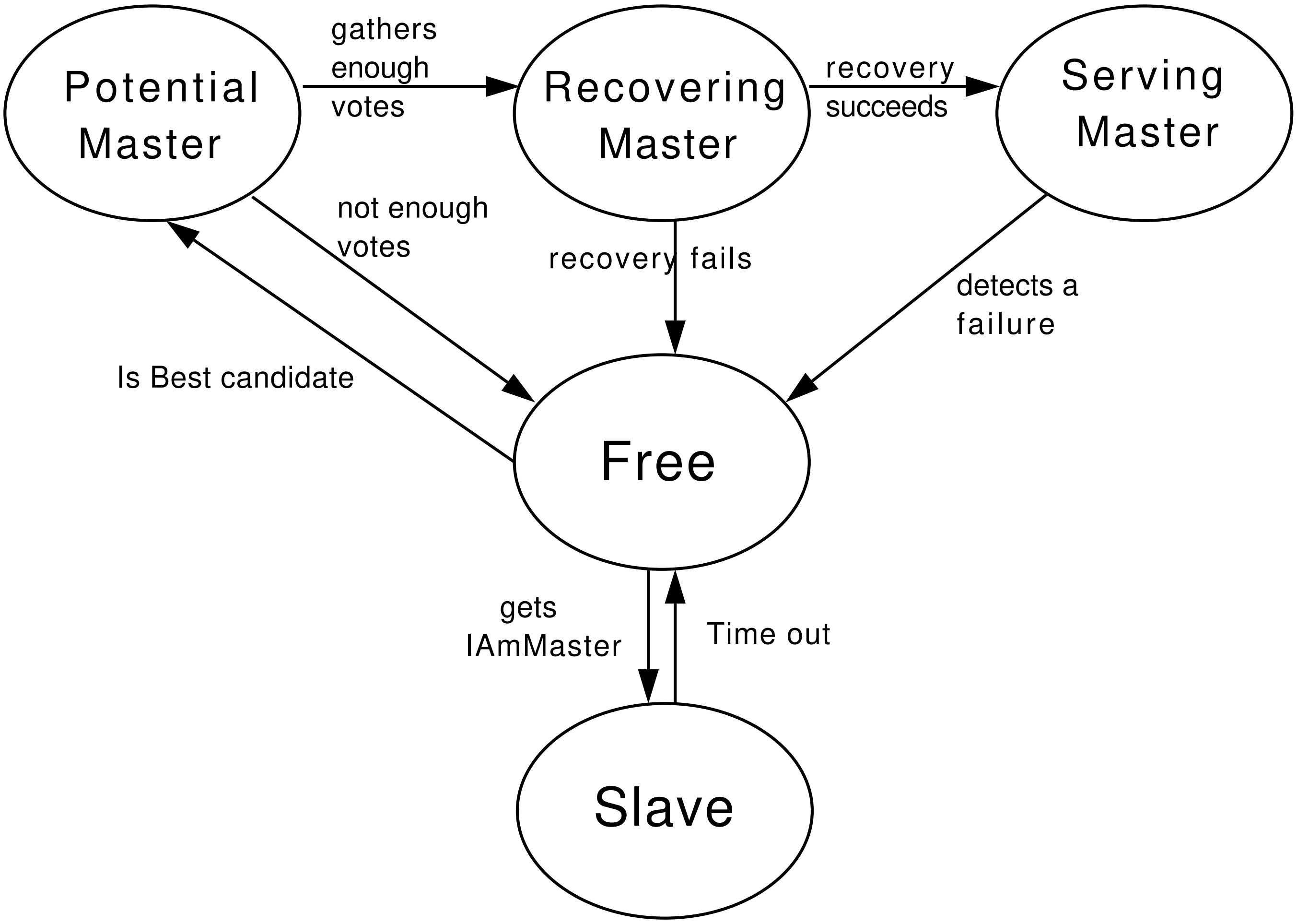

Figure 4: Replica states

free replica moves into the **Slave** state if it receives an IAmMaster call from another replica.

- **Slave**. A replica in this state has agreed to be a slave to another replica. When a replica agrees to be a slave, its agreement lasts for *MasterTimeout* seconds as measured on its own interval timer, where *MasterTimeout* is a value known to both the slave and its master. If the agreement expires without having been renewed by the master, the replica reenters the **Free** state. A slave must be sure not to violate its agreement even if it crashes.[5]

- **PotentialMaster**. A replica in this state is busy collecting slaves, trying to become master. It enters the **RecoveringMaster** state if it succeeds, the **Free** state if it fails.

- **RecoveringMaster**. A replica in this state has convinced a majority of the replicas to be its slaves. It is now the one and only master. While in this state, the master directs the recovery phase of the `Ar` algorithm. It moves to the **ServingMaster** state if recovery succeeds, the **Free** state if it fails.

- **ServingMaster**. A master replica in this state is ready to provide service. It returns to the **Free** state if it detects a failure.

The overall system is thus in the **Service** phase if there is any replica in the **ServingMaster** state, in the **Recovery** phase if there is a replica in the **RecoveringMaster** state, and otherwise in the **Election** phase.

2.2 Service

During normal service, the clerks pass on to the master all read and write requests that they cannot handle using their local caches. The master handles read requests simply by reading its own stably-stored copy of the array and returning the value to the requesting clerk. When the master receives a write request, it assigns the request a sequence number, then passes the request on to all the active replicas in parallel (including itself). The replicas respond to the master only after they have recorded the write on stable storage, and the master responds to the clerk only after all replicas have responded.

[5] Therefore, when a crashed replica comes back up, it remains dormant for *MasterTimeout* seconds before beginning to run the `Ar` algorithm.

The master and slaves use multiple threads of control to accept read and write requests, so at any moment several may be in progress concurrently. We require the results of concurrent `Ar.Get` and `Ar.Set` operations to be the same as if the operations were serialized in some total order consistent with the partial orders seen by each client.[6] Operations that overlap in time can be serialized in any order; in other words, when a call to `Ar` returns, the caller knows only that the results reflect the state of the array at some time during the interval between the call and the return.

To improve performance during the **Service** phase, the master does not use an atomic commit protocol to forward writes to its slaves; it simply makes a remote procedure call to each one and waits for them to respond. We say a write is *committed* when it is stably recorded by the current master and all its slaves. A committed write is never undone. Clearly, if a crash occurs while writes are in progress, some replicas may have one or more uncommitted writes recorded on their stable storage when the system enters the **Election** phase. Uncommitted writes are *reconciled*—either committed or backed out on all replicas—during the **Recovery** phase. (The choice to commit or back out is made nondeterministically.) To support reconciliation, each replica must know which of the writes it has carried out may be uncommitted, and must have logged enough information to back them out locally or push them forward on other replicas that have not seen them. To limit the space taken up by logs, we limit the number of concurrent writes that the master may have in progress during the **Service** phase to an agreed-on value called *NInProgress*. The master blocks requests with sequence number s or greater until all requests numbered $s - \mathit{NInProgress}$ or earlier have completed.

Before a master can safely handle a read request, it must make sure that it is *still* the master—that it is in fact recognized as master by a majority of the replicas. To permit this without requiring the master to contact its slaves on every read request, we use *slavery agreements* that are valid for a known length of time. When a replica becomes a slave, it agrees to remain enslaved to the same replica for at least *MasterTimeout* seconds, a time known to both the master and slave. Periodically throughout the **Recovery** and **Service** phases, the master calls each slave to renew its agreement. Just before the master responds to a read request, it reads its local clock and verifies that its slavery agreements are all still valid as of

[6] In fact, `Ar.Set` operations that reach the master and become committed are serialized in the order of their sequence numbers.

that time.[7] Thus, the result it returns is correct as of that time.

The system leaves the **Service** phase and enters the **Election** phase whenever there is a change to the set of replicas that the master is able to exchange messages with. How are such changes detected? As just described, the master periodically calls its slaves to renew their agreements, and of course it also calls them frequently to propagate writes. If any of these calls fail, the master stops providing service, enters the **Free** state, and begins running the election algorithm. In addition, the master periodically attempts to call each of the replicas that is not currently its slave. If any of these calls succeed, again the master enters the **Free** state and begins a new election. Finally, whenever a replica's slavery agreement times out without having been renewed, that slave enters the **Free** state and begins running the election algorithm.

The length of time for which slavery agreements are renewed by each call (*MasterTimeout*) must be chosen carefully. Increasing the timeout causes the master to poll less often, reducing the load on the servers and the network, but it increases the time needed for the slaves to detect that the master has failed and elect a new one. We believe a timeout of about one second will be appropriate for our file system.

Pseudo-code for the Service phase. The following pseudo-code summarizes the master's operation during the **Service** phase. All the variables mentioned in the code are local to the master. The set of active replicas (the master and its slaves) is called *Active*, while *Replicas* is the set of all replicas, active or not. Two other sets are also used in the PreserveAgreements code: *Con* and *Slaves*. *Con* is the set of replicas that this replica succeeded in contacting on its last try, while *Slaves* is the set of replicas that are its slaves (including itself). In the **Service** phase both these sets are equal to *Active*, but the PreserveAgreements process also runs during other phases of the algorithm.

As stated in Section 1.4 (safety condition S2), we assume the difference in running rate between any replica's clock and real time is bounded by a small constant ρ. Violation of this bound can result in incorrect operation: replica p may believe that replica q is its slave, while at the same time replica q, believing its agreement with p has expired, takes on a different master. The code also uses γ as a bound on the time for the master to make a remote

[7] Note that the master must allow for the possibility that the slaves' clocks have been running slightly faster than its own, within the bound ρ defined in Section 1.4.

procedure call on a slave, but this bound is not necessary for correctness; if it is violated, at worst an unnecessary election is triggered.

```
procedure Read(i)
    (* Called remotely by clerks *)
    if not StillMaster() then
        RestartElection();
        raise exception NotMaster;
    end if;
    return Ar[i];
end Read;

procedure Write(i, val)
    (* Called remotely by clerks *)
    if not StillMaster() then
        RestartElection();
        raise exception NotMaster;
    end if;
    parfor s ∈ Active do
        try
            SlaveWrite(s, i, val);
        on exception
            RestartElection();
            raise exception Failed;
        end try;
    end parfor;
end Write;

procedure StillMaster
    t := current time on local clock;
    for s ∈ Slaves do
        if slaveryEnds[s] < t then return false;
    end for;
    return true;
end StillMaster;


procedure RestartElection
    myState := Free;
```

```
        stop process PreserveAgreements;
        stop process Recovery;
        start process Election;
end RestartElection;


process PreserveAgreements
        loop
                t := current time on local clock;
                c := t + (MasterTimeout · (1 − 2ρ));
                parfor s ∈ Slaves do
                        try
                                RenewAgreement(s, MasterTimeout);
                        on exception
                                RestartElection();
                                exit process;
                        end try;
                        slaveryEnds[s] := c;
                end parfor;
                parfor s ∈ (Replicas − Con) do
                        if AreYouThere(s) then
                                RestartElection();
                                exit process;
                        end if;
                end parfor;
                pause until time c − γ;
        end loop;
end PreserveAgreements;
```

2.3 Election

In the **Election** phase, the replication algorithm's goal is to choose a master that is up, up-to-date, and able to gather at least a majority of all the replicas (counting itself) as slaves. A replica is *up-to-date* if it has applied every committed write operation to its stably-stored copy of the array. (As was explained in Section 2.2, a write operation becomes committed if either (1) it completes successfully during a **Service** phase, or (2) it is started but not completed during a **Service** phase, and a subsequent **Recovery** phase decides to commit it as part of reconciliation.)

There are three reasons why we require the elected master to be up-to-date. First, since the master must be up-to-date to do its job in the **Service** phase, the system can go into service more quickly if it elects a master that is already up-to-date than if it must take the time to bring the master up-to-date. Further, if the master is known to be up-to-date while it is directing reconciliation, the other replicas can simply update themselves from it. Finally, when we extend the algorithm to allow changing the replica set (Section 2.5), it will be convenient to require that the master's view of the replica set be up-to-date. Doing so permits us to reconcile the sets seen by the various replicas by imposing the master's view on all of them, and also makes it easy to ensure that a replica that has been deleted from the set is not elected master.

Epochs. Our algorithm determines which replicas are up-to-date using a set of four *epoch* variables kept in stable storage on each replica.[8] The epochs are examined during **Election** to find an up-to-date master, then advanced during **Recovery** to show which replicas are up-to-date as of the new epoch. Before going into the other details of the **Election** and **Recovery** phases, we give a general explanation of how epochs work in the next few paragraphs.

The most basic of the four epochs is the *service* epoch. During recovery, after the active replicas have reconciled their copies of the array so they are all up-to-date, they advance their service epochs to a new, common value, larger than any that has ever been used before. Meanwhile the service epochs of the inactive replicas remain as they were, so the active replicas are now marked as having more recent data. The set *Active* is required to be a majority, and entry to the **Service** phase fails (triggering a new election) if any member of *Active* is unable to advance its epoch.

Now, consider the following proposition $\mathcal{P}$:

> Let M be any set containing a majority of the replicas, and let $maxService(M)$ be the largest service epoch of any replica in M. Then every replica r with $service(r) \geq maxService(M)$ is up-to-date.

It is easy to see that this proposition is true immediately after a transition to the **Service** phase with $Active = A$: First, the sets M and A must

[8] The epoch variables themselves are not in any sense "replicated"; each replica has its own epochs, applying only to itself.

overlap, because both are majorities; further, each member of A is up-to-date; and finally the members of A have larger service epoch numbers than any other replica. So any replica with $service(r) \geq maxService(M)$ must be a member of A, and hence up-to-date.

Moreover, $\mathcal{P}$ remains true throughout the **Service** phase and the following **Election** phase, because the variables it talks about—service epochs and which replicas are up-to-date—do not change during these phases.

In fact, $\mathcal{P}$ remains true at all times, even when a **Recovery** phase changes some replicas' service epochs but fails to bring the system into the **Service** phase. Suppose that in some election, $\mathcal{P}$ is true and is used to choose an up-to-date replica r. Before the **Recovery** phase alters any replica s's service epoch, it first brings s up-to-date (by reconciling it with r); it then *increases* s's service epoch to the new value selected for this recovery. This action obviously cannot make any replica other than s appear up-to-date according to $\mathcal{P}$ (since it cannot reduce $maxService(M)$ for any M), and it is correct for it to make s itself appear up-to-date. So once $\mathcal{P}$ has become true, it remains true every step of the way, regardless of whether recovery succeeds or fails. ($\mathcal{P}$ is true when a new system is first turned on because at that time all replicas are up-to-date.) Therefore the **Election** phase can always use $\mathcal{P}$ to choose an up-to-date master.

We have glossed over one point, however—how do the replicas choose a new epoch value larger than any that has ever been used before? It is not sufficient for the new value to be larger than what was used for the last *successful* recovery. Like data writes, new epoch values are passed from master to slaves non-atomically, so a failure during recovery can leave a minority of replicas with the new service epoch—call it e. A later election could then gather a majority that does not include any of those replicas. Yet a new epoch larger than e must be chosen for the subsequent recovery, or the failed replicas will falsely appear up-to-date if they are up during the next election.

To solve this problem, we introduce a second epoch variable on each replica, called the *big* epoch. Choosing a new epoch then becomes a two-phase process. First, the master computes $newEpoch = \max\{big(r) : r \in Active\} + 1$. Then, for each replica r in $Active$, the master sets $big(r)$ to $newEpoch$. Only after this succeeds on all replicas in $Active$ does the master begin advancing the service epochs, using $newEpoch$ as the new value. Thus the new value, though it may be less than $big(s)$ for some replica s not in $Active$, is larger than $service(s)$ for any replica s.

Our epoch algorithm is now correct, but recovery is too slow. With the algorithm as described so far, a slave's data must be fully reconciled with the master's before the slave's epoch can be updated, and all slaves must be updated before the system can go into service. We would prefer an algorithm that brings the system into service more quickly, reconciling slaves that are far behind the master in the background, while service is being provided. This is not important for an array of 100 integers, of course, but we are looking ahead to the needs of a real file system with gigabytes of on-line data, where reconciling a replica that is far behind the master could take many minutes.

We can make this improvement by introducing a third epoch variable, the *data* epoch. A replica r's data epoch holds the epoch value of the most recent **Service** period in which r was up-to-date, while r's service epoch holds that of the most recent **Service** period in which r was active. During recovery, if a slave is found to be far out-of-date, its reconciliation is handled by a separate process that is forked from the main thread of recovery. The slave's service epoch is set to *newEpoch* when the master takes the system into the **Service** phase, but its data epoch is advanced only after its reconciliation is complete.

With this change to the epoch scheme, the eligibility test for masterhood becomes more complicated. In place of just picking the largest service epoch of a majority, we must require that the master r have $data(r) = service(r)$ as well, so that replicas that crashed while being reconciled are excluded. Note that, although there is always at least one up-to-date replica in the system (the last replica to have been master, for instance), it is now possible for a majority of the replicas to be up and connected, but for none of them to be up-to-date. In this case, no master can be elected until an up-to-date replica joins the connected majority. Thus, adding the data epoch has reduced the algorithm's fault tolerance in return for faster recovery.

The data epoch has one additional benefit; it allows us to introduce *witnesses* into the replica set [23]. A witness is a replica that does not keep a copy of the array, but participates in the election algorithm to break ties. Such replicas have a data epoch of zero at all times, and writes are not sent to them. A system configuration containing only two replicas becomes feasible with the addition of a witness.[9]

One problem remains with the algorithm as described so far—faults during recovery can reduce the set of replicas that appear up-to-date and are

[9] A witness replica does not require a dedicated host machine; it can be run on a host that is also a platform for other services.

eligible to be master. Suppose there is a crash while the service epochs are being advanced to a new value e, so that the epochs of some replicas are advanced while others are not. If the participants in the next election include any replicas whose epochs were advanced to e, only those replicas can be eligible for masterhood—the others will appear out-of-date, even though no writes were actually performed during epoch e. Now, if all the replicas with service epoch e are witnesses (or out-of-date replicas with $data(r) < service(r)$), no master can be elected until a replica r with $data(r) = service(r) = e$ becomes connected to the majority. To avoid permanent blockage, we must ensure that such a replica always exists. One way to do so is to insist that a master running recovery advance the service and data epochs of the up-to-date replicas before it advances the service epoch of the out-of-date replicas and witnesses. This algorithm still has a drawback, however—in the next recovery, replicas that appear out-of-date but are in fact up-to-date must undergo a slow background reconciliation rather than the relatively fast reconciliation available to up-to-date replicas.

An alternative approach to the problem of faults during recovery involves introducing a fourth epoch variable, the *prospective* epoch. The idea is that the master advances each active replica's prospective epoch to $newEpoch$ early in recovery, immediately after choosing $newEpoch$. Then any replica r with $prospective(r) = e$ must have been active during recovery number e, so if $service(r) < e$, recovery e must have failed, and the existence of another replica s with $service(s) = e$ does not prevent r from becoming master. With one more refinement, this idea yields a correct algorithm. While the replicas are advancing the prospective epochs, if any replica r previously had $prospective(r) < maxService$, it marks itself as being out-of-date by setting $service(r)$ to equal $maxService$ (to make $service(r) > data(r)$) before setting $prospective(r)$ to $newEpoch$. In this final version of the algorithm, a replica r is up-to-date (and hence eligible for masterhood) if (1) $data(r) = service(r)$, and (2) $prospective(r) \geq \max\{service(s) : s \in M\}$, where M contains a majority of all the replicas.

The election algorithm. Whenever a replica r enters the **Free** state, it begins running the election algorithm, which goes as follows.

First, r tries to contact every replica in the replica set. Each replica that r can contact returns a snapshot of its epochs, plus its state and (if it is a slave) the name of its master. As long as r remains in the **Free** state, it periodically calls every replica to update this snapshot. Let $Con(r)$

be the set of replicas that responded the last time r called them (initially empty). For a short time after r first enters the **Free** state, $Con(r)$ will be changing rapidly, as r contacts each replica for the first time. But if r waits long enough and if no changes in network or replica up/down status occur, $Con(r)$ will stabilize. So, as a heuristic to avoid doing extra work, r waits until $Con(r)$ has remained unchanged for a while before going on to the next step.[10]

Next, after $Con(r)$ appears to have stabilized, r decides whether it should nominate itself as master. It does so if:

1. $Con(r)$ contains a majority of the replicas,

2. No member of $Con(r)$ is a slave to any replica other than r,

3. According to the epochs r has gathered, r appears to be up-to-date, and

4. Of the replicas that appear to r as up-to-date, r itself is the "best"—that is, the value $goodness(r)$ is larger than $goodness(s)$ for any other apparently up-to-date replica s.

Items 3 and 4 require some additional explanation. To determine whether a replica s is "apparently up-to-date," r uses the state snapshots it has gathered for the replicas in $Con(r)$ to evaluate the up-to-date predicate discussed above, with $M = Con(r)$. That is, s appears up-to-date to r if (1) $Con(r)$ contains a majority of the replicas, (2) $data(s) = service(s)$, and (3) $prospective(s) \geq \max\{service(t) : t \in Con(r)\}$, where all the epoch values mentioned are the values in r's snapshot, not the actual values on the replicas s and t. To determine which master candidate is best, the replicas use an arbitrary, prearranged function. For example, if each replica has a unique numeric identifier, $goodness(r)$ can simply be r's identifier. Thus, ordinarily, if a majority of replicas are up and at least one is up-to-date, exactly one replica will determine that it should nominate itself as master.[11] A replica that does not decide to nominate itself as master remains in the **Free** state and goes back to gathering snapshots, until it either receives new

[10] The exact time to wait is arbitrary; it should be a bit more than the maximum time usually needed to contact a replica that is up and has a working network connection to r.

[11] Some combinations of faults can cause more or fewer replicas to nominate themselves, but the algorithm remains safe if this occurs, and it does not occur if the progress conditions are met throughout the election.

information that allows it to nominate itself, or receives an IAmMaster call making it a slave to another replica.

When a replica r does decide to nominate itself as master, it enters the **PotentialMaster** state. On entry to this state, r picks a *master incarnation number* (unique and increasing with time across all transitions to the **PotentialMaster** state by r).[12] Next, r makes an IAmMaster call on each of the replicas in $Con(r)$ in parallel, asking them to be its slaves, and providing its incarnation number and its prospective epoch. In a background task, r periodically renews the agreements with the slaves it has successfully gathered, and periodically attempts to contact the replicas that are not in Con, just as it would in the Service state. If any replica refuses an IAmMaster call, any slave does not respond to a renewal call, or any replica not in Con is contacted, r reenters the **Free** state and restarts the election algorithm.

How does a replica s decide whether to accept an IAmMaster call from r? First, s must either be in the **Free** state, or be a slave to r already (with an incarnation number less than or equal to the one given in the call). Second, $prospective(r)$ as given in the call must be greater than or equal to $service(s)$. If both these tests succeed, s accepts the call—it remembers the caller's identity and incarnation number, changes its state to **Slave** and responds affirmatively. If either test fails, s refuses the call—it leaves its state unchanged and responds negatively. The first test ensures that slaves do not violate their agreements, and that a potential master or established master cannot be enslaved until it determines for itself that it has failed to become (or remain) master. The second test compares r's prospective epoch against the actual service epoch of s, rather than just against r's snapshot. This test is necessary because $service(s)$ could have changed since r's snapshot was taken—but once s agrees to be r's slave, it will no longer change its epochs except at r's request.

If all the members of $Con(r)$ respond affirmatively, r examines its local clock and data structures to be sure its slaves' agreements are all still valid (they might not be if r has been running very slowly), then enters the **RecoveringMaster** state if they are. Since all members of $Con(r)$ have responded affirmatively and have become r's slaves, r is now certain that

[12] This incarnation number (together with r's identity) is used by r's slaves to detect and reject delayed messages from previous masters, including previous incarnations of r. If r ever receives such a rejection, it returns to the **Free** state and restarts the election algorithm. The generation and checking of incarnation numbers is not shown in the pseudo-code, but should be understood as present.

$prospective(r) \geq service(s)$ for all s in $Con(r)$, so r is definitely up-to-date (not just apparently up-to-date). However, if any of the agreements are no longer valid, r returns to the **Free** state and restarts the election algorithm from the beginning.

Pseudo-code for the Election phase. The following pseudo-code summarizes the **Election** phase of the `Ar` replication algorithm. All the variables mentioned in the code are local to the replica running it, which we assume is named r. That replica's own, locally-stored epochs are denoted $data(r)$, $service(r)$, etc., while its snapshot values of another replica's epochs (say, s's) are denoted $data(s)$, $service(s)$, etc.

```
process Election
    (* Gather snapshots *)
    Con := ∅;
    repeat
        oldCon := Con;
        pause for time ConStableTime;
        parfor s ∈ Replicas do
            try
                GetSnapshot(s);
                Con := Con ∪ {s};
            on exception
                Con := Con − {s};
            end try;
        end parfor;
    until Con = oldCon and |Con| > |Replicas|/2;

    (* Check if best candidate for master *)
    maxService := max{service(s) : s ∈ Con};
    if
        No s ∈ Con is enslaved to a replica other than r and
        data(r) = service(r) and
        prospective(r) ≥ maxService and
        There is no s ∈ Con such that
            data(s) = service(s) and
            prospective(s) ≥ maxService and
            goodness(s) > goodness(r)
    then
```

```
            myState := PotentialMaster;
        else
            RestartElection();
            exit process;
        end if;

        (* Try to become master *);
        Slaves := ∅;
        start process PreserveAgreements;
        parfor s ∈ Con do
            try
                t := current time on local clock;
                c := t + (MasterTimeout · (1 − 2ρ));
                IAmMaster(s);
                slaveryEnds[s] := c;
                Slaves := Slaves ∪ {s};
            on exception
                RestartElection();
                exit process;
            end try;
        end parfor;
        if StillMaster() then
            Active := Slaves;
            myState := RecoveringMaster;
            start process Recovery;
        else
            RestartElection();
        end if;
    end Election;
```

2.4 Recovery

In the **Recovery** phase of the `Ar` algorithm, the master directs the replicas in two intertwined tasks. The first task is to update the epoch variables so that all replicas that might be out-of-date, including those that are down, can determine that they are out-of-date in the next election. The second task is to eliminate (or *reconcile*) any differences between the arrays stored by up-to-date replicas. Figure 5 illustrates how these tasks are intertwined: in-

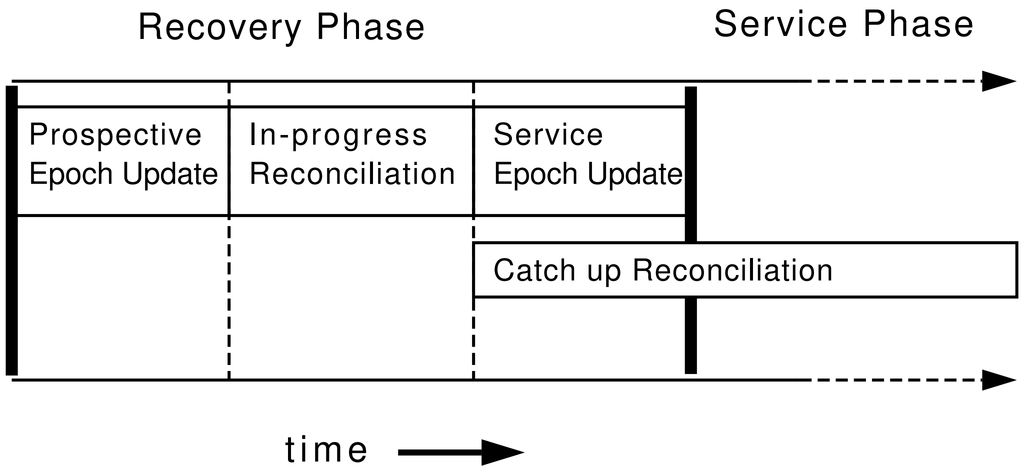

Figure 5: The Tasks in Recovery

progress reconciliation occurs between prospective and service epoch update, while catchup reconciliation begins concurrently with service epoch update and can continue beyond it into the **Service** phase.

Throughout the recovery phase, the recovery fails if the master cannot contact any slave it is trying to call, if any slave's agreement expires, or if the master succeeds in contacting a replica that is not its slave. The master stops running the recovery algorithm immediately, reenters the **Free** state, and restarts the election algorithm.

The first step in recovery marks each active replica that may be out-of-date, so that later steps cannot make it falsely appear up-to-date. The master r computes the value $maxService = \max\{\mathit{service}(s) : s \in \mathit{Active}\}$, then calls each slave s for which $\mathit{prospective}(s) < maxService$, instructing it to atomically advance its prospective and service epochs to equal $maxService$, but to leave its data epoch unchanged (and thus less than $maxService$). Now s can tell it is out-of-date without looking at the epochs of other replicas.

Second, the master chooses a new epoch value to use for the next **Service** phase. As described above in our discussion of epochs, r computes $\mathit{newEpoch} = \max\{\mathit{big}(s) : s \in \mathit{Active}\} + 1$, and then asks each active replica (including itself) to set its big epoch to $\mathit{newEpoch}$. If this succeeds, $\mathit{newEpoch}$ is an epoch value that has never been used before (and will never be used again) for the next recovery step.

In the third step of recovery, the master asks each active replica to set its prospective epoch to *newEpoch*. These replicas are now all marked as having participated in this recovery.

Next, the master begins the reconciliation process. We describe only *what* reconciliation does at this point, deferring discussion of *how* it is done to the end of this section.

As the fourth step of recovery, the master performs *in-progress reconciliation*—it reconciles the copies of the replicated data held by it and by all the other up-to-date replicas (replicas s for which $data(s) = service(s)$ and $prospective(s) \geq maxService$). This process is called *in-progress* reconciliation because the up-to-date replicas cannot disagree on anything but those operations that were still in progress when the last service period ended; they are otherwise identical. After in-progress reconciliation, the up-to-date copies are completely identical.

As the fifth recovery step, the master starts a background process to reconcile each out-of-date replica, modifying its data copy to match the master's copy. We call this process *catchup* reconciliation, because the out-of-date replica is catching up on operations that were done during service periods when it was inactive.[13]

The sixth and final recovery step advances the active replicas' service epochs. The master asks each up-to-date replica to (atomically) advance its data and service epochs to *newEpoch*, while asking each replica that is still doing catchup reconciliation to advance only its service epoch, leaving its data epoch unchanged. Such a replica advances its data epoch when it has completed catchup reconciliation (having thus been brought up-to-date).

If all these steps succeed, the master enters the **ServingMaster** state, and a new **Service** phase begins.

In-progress reconciliation. Looking ahead to the real file system implementation, we have decided to do in-progress reconciliation using logs. Each replica keeps on its log a stable record of all the write operations it has performed that might still be in progress. For each operation, enough information is logged so that the replica can choose between undoing the operation or passing it to the other replicas to be done. The master numbers the write operations as it receives them, and these numbers are also logged.

[13]The algorithm allows catchup reconciliation to be done in the background because it is likely to take a long time on a real file system; on the `Ar` prototype, of course, it is practically instantaneous.

To perform in-progress reconciliation, the newly elected master gives each slave a list of the operations in the master's log for the last service phase, identified by number. Slaves that have done operations not on the list undo them in reverse order. Slaves that have missed operations on the list request their operands from the master and do them.

To keep the logs from growing too large, we place a limit on the number of operations that can be in progress concurrently, called *NInProgress*. The master is not permitted to issue operation k until the operations numbered $k - NInProgress$ and earlier have completed on all replicas. Thus, at most the last *NInProgress* operations need be kept in each replica's log. *NInProgress* should be chosen large enough that the master may issue enough concurrent writes to the replicas to keep them busy, but not so large that the replicas run out of log space.

We believe that a log-based approach like this one is highly appropriate for implementing a file system, independent of its usefulness for reconciling replicated data. It is widely recognized that disks have far higher performance when used as sequential-access logging devices than when used as random-access devices [11]; or for still better performance, battery-backed semiconductor memory can be used as a logging device. The file system can thus make writes stable with low latency, though its throughput is still limited by the speed with which data can be written to its permanent location.

Many variations are possible in the details of this reconciliation scheme. For example, we could avoid the need for undo by rolling both the master and the slaves forward, so that any operation that was seen by any replica in the last service period is now done by all. The exact scheme to be used in Echo remains to be decided at this writing.

Catchup reconciliation. We are aware of three possible methods for doing catchup reconciliation: full-copy, compare-and-copy, and log-based. We merely sketch the methods here, because `Ar` does not provide a good framework for describing them; they all make much more sense in the context of a real file system.

In the *full-copy* method of catchup reconciliation, the data held by the out-of-date replica is thrown away and replaced by a copy of the master's data. This can of course be done quickly in the `Ar` prototype, but in a real file system the large volume of data would make this approach slow. Nonetheless, we plan to use it to update replicas that are extremely far out of date, or that must start from scratch because of a disk head crash or the like. It is somewhat tricky to do a full-copy in background while the master is providing service and modifying its copy of the data. The slave must receive the updates that are being done during the service period and save them, so that it can apply them to its copy after it has finished making it. Moreover, the slave must perform the copying process in a way that leaves it with a well-formed data structure to which those updates can be applied to yield the correct result, even though the original is changing during the process.

In the *compare-and-copy* method of catchup reconciliation, the out-of-date replica compares its data with that held by the master, and copies over those parts that differ. When the replicated data is a tree of files and directories, the comparison can be done quickly by walking the tree and comparing time stamps (or version stamps) on corresponding files. This technique is faster than full-copy if the comparison process is fast and the amount of data that needs to be copied is not too large. As with full-copy, if a slave is to do a compare-and-copy in the background while the master is providing service, it must save the updates the master performs during the process and apply them to its copy before going into service.

The *log-based* method of catchup reconciliation is similar to in-progress reconciliation. Each replica keeps write operation logs going as far back as possible, considering the amount of storage available. When an out-of-date slave is to be reconciled, it examines its own log and the log of its master to determine what operations it must undo from the last service period in which it was active, and what operations the master has done since. It then performs the necessary undo and redo operations.[14] If the slave has been inactive for too long, the master's log may not go back far enough for log-based reconciliation to work; in that case the system falls back on full-copy or compare-and-copy. We are uncertain whether this method offers a worthwhile performance improvement over compare-and-copy. At present we are hoping it will not be necessary to implement it.

Note that if a new election is triggered while a replica is engaged in catchup reconciliation, the catching-up replica will play a role in the election

[14] Alternatively, the slave could examine the logs and note which portions of the replicated data were touched by these operations and copy them over, as in the compare-and-copy method. This technique does not get any data values from the logs, only the names of the portions that were affected by the operations, so one can *compress* old logs that are being kept only to support catchup reconciliation, removing the data values to save space. This technique also has the advantage that undo is not required, which is significant in the file system case as old values of files (which can be large) need not be logged.

that resembles that of a witness replica. It is not eligible to become master (its service epoch is greater than its data epoch) but it can serve as a vote in forming a majority.

Pseudo-code for the Recovery phase. The following pseudo-code summarizes the recovery algorithm as run by a master replica r. For brevity, the details of calling slaves over the network and handling failures are omitted. Whenever the pseudo-code shows an assignment to one of a slave's epoch variables, or a write operation being done or undone on a slave, the master must actually call the slave, asking it to make the change.[15] If any such call fails, the master immediately calls RestartElection, stopping the Recovery process. The PreserveAgreements process continues to run in the background during recovery.

```
process Recovery
    maxService := max{service(s) : s ∈ Active};

    (* Mark out-of-date replicas *)
    parfor s ∈ Active do
        if prospective(s) < maxService then
            atomic
                prospective(s) := service(s) := maxService;
            end atomic;
        end if;
    end parfor;

    (* Choose a new epoch value *)
    newEpoch := max{big(s) : s ∈ Active} + 1;
    parfor s ∈ Active do
        big(s) := newEpoch;
    end parfor;

    (* Advance prospective epoch *)
    parfor s ∈ Active do
        prospective(s) := newEpoch;
    end parfor;
```

[15]The master need not call its slaves to read their epoch variables, because the snapshots it has left over from the **Election** phase remain accurate as long as the slaves' agreements remain valid.

```
    (* Perform in-progress reconciliation *)
    masterOps := sequence of last NInProgress operations
        done by the master;
    parfor s ∈ Active do
        if data(s) = service(s) then
            slaveOps := sequence of last NInProgress operations
                done by s in the last Service phase;
            for op := LAST(slaveOps) downto FIRST(slaveOps) do
                if op ∉ masterOps then undo op on s; end if;
            end for;
            for op := FIRST(masterOps) to LAST(masterOps) do
                if op ∉ slaveOps then do op on s; end if;
            end for;
        end if;
    end parfor;

    (* Start catchup reconciliation *)
    parfor s ∈ Active do
        if data(s) ≠ service(s) then
            Start catchup reconciliation on s;
        end if;
    end parfor;

    (* Advance service epoch *)
    parfor s ∈ Active do
        if data(s) = service(s) then
            atomic
                service(s) := data(s) := newEpoch;
            end atomic;
        else
            service(s) := newEpoch;
        end if;
    end parfor;

    myState := ServingMaster;
end Recovery;
```

2.5 Changing the replica set

In this section, we describe an extension to the `Ar` algorithm that permits a system administrator to safely change the `Ar` replica set whenever the system is in the **Service** phase. This facility is useful for configuring out replicas that have been destroyed permanently or are no longer needed, to configure in additional replicas for improved availability, or to move replicas from one place to another.

The extended algorithm has the same safety and progress conditions as the original one. In particular, as long as the safety conditions are met, the system cannot fission into two independent sets of replicas both providing service, even if faults occur during execution of a replica set change. Also, the system cannot get into a state where it is permanently unable to provide service: even if there are faults during a replica set change, the system will provide service again once the progress conditions are again met. As with write operations, if faults occur during a replica set change, the next recovery restores consistency: either the attempted change is carried through on all active replicas and becomes stable, or it is backed out and has no effect. Inactive replicas are brought into agreement with the majority view during the next recovery in which they become active.

The operations we allow on the replica set are additions and deletions of one replica at a time. (This restriction is important for the correctness of our algorithm, because it ensures that any majority formed before the change must intersect with any majority formed after the change.) The interface used for manipulating the replica set is given in Figure 6. Note that the replicas are named by text strings; we assume there is an underlying name service that provides a way of mapping between these text strings and network addresses.

For convenience, this interface is exported to clients by the `Ar` clerks. A clerk receiving one of these requests simply passes it on to the master, with no local processing. With this arrangement, all communication between clients and the `Ar` service is through the clerks, so clients are insulated from the details of locating the current master (discussed in the next section).

How is this extension to the `Ar` algorithm implemented? Besides the addition of the DeleteReplica, AddReplica, and GetReplicaSet procedures themselves, three changes are needed to the algorithm as given so far.

First, throughout the extended algorithm, each replica r keeps its own notion of the set of replicas on stable storage; we denote this set as $Replicas(r)$. Wherever the basic algorithm uses the set $Replicas$, the ex-

```
SAFE DEFINITION MODULE ArConfig;

FROM Ar IMPORT ServiceUnavailable;

IMPORT Text;

TYPE
  SetOfReplicas = REF ARRAY OF Text.T;

PROCEDURE DeleteReplica(victim: Text.T)
  RAISES {ServiceUnavailable};

PROCEDURE AddReplica(newcomer: Text.T)
  RAISES {ServiceUnavailable};

PROCEDURE GetReplicaSet(): SetOfReplicas
  RAISES {ServiceUnavailable};

END ArConfig.
```

Figure 6: The `ArConfig` interface.

tended algorithm uses $Replicas(r)$, where r is the replica running the code in question.

Second, in the **Recovery** phase of the extended algorithm, as part of reconciliation, the master r calls each slave s, asking it to set $Replicas(s)$ equal to $Replicas(r)$. As usual, if this step fails, r immediately reenters the **Free** state and restarts the **Election** phase.

Third, in the **Election** phase, the criteria that a replica r uses to decide whether it should nominate itself as master are augmented. The first four criteria are as before—$Con(r)$ must be a majority, $Con(r)$ must not contain any members that are slaves to a replica other than r itself, r must appear up-to-date, and r must have the highest *goodness* value of all replicas in $Con(r)$ that appear up-to-date. But in addition, if more than one replica appears up-to-date, r must be a member of $Replicas(s)$, where s is the up-to-date replica with the second-highest goodness value. (The value of $Replicas(s)$ is obtained from r's snapshot of s's state.) This added criterion is needed to ensure progress in the situation where there is a crash during the deletion of the replica r with the highest *goodness* value, after the replica s with the second highest value has heard about the deletion, but before r itself has heard about it. In this situation, without the added criterion, both r and s would try to become master, and it is quite possible that neither would ever succeed, even in the absence of further faults.[16] It is sufficient for r to check only the second-best alternative candidate because deletions are done one at a time.

GetReplicaSet is trivial to implement; the master r simply returns its set $Replicas(r)$.

DeleteReplica and AddReplica are implemented as follows. For both, the master r first stops accepting new Write requests and waits until all in-progress Write operations are complete. Next, r calls every replica in *Active* (plus the replica being added, if this is AddReplica), giving each the new value of the replica set. If the operation is DeleteReplica(t), r does not call t until all the other calls have returned; otherwise r can make the calls in parallel. Finally, the master r enters the **Free** state and begins running the election algorithm (unless it has just deleted itself). As usual, if any of the remote calls in this procedure fail, or if the master succeeds in contacting an inactive replica while the procedure is running, the master immediately stops running it, enters the **Free** state, and begins running the election algorithm.

A new replica r that has not yet been added to the replica set is initialized with $big(r) = prospective(r) = service(r) = 0$, $data(r) = -1$, and $Replicas(r) = \emptyset$, making it ineligible to become master. Its replica set and epochs are advanced to the current values when the master executes the AddReplica procedure and the subsequent Recovery process.

A replica r that has been deleted from its own set $Replicas(r)$ can turn itself off; its deletion will never be backed out by recovery, because every active replica has heard about it. It is harmless for such a replica to keep running until it is turned off by hand, however; it cannot become master because, not being in $Replicas(r)$, it will never enter $Con(r)$ and thus cannot become the best candidate in $Con(r)$.

Note also that a replica r that is deleted while it is inaccessible to the master is never told about the deletion. A system administrator would ordinarily turn r off by hand after deleting it, but again, it is harmless if r keeps running—r is out-of-date and can never become master itself

[16] Of course, they could not both succeed, so the criterion is not needed for safety.

(though it might expend effort gathering the set $Con(r)$), and every up-to-date replica s will ignore r when trying to gather its own $Con(s)$.

The following pseudo-code summarizes the extensions just described. We assume the replica running this code is named r. When the pseudo-code shows an assignment to another replica s's local state, this means that r calls s, requesting it to make the change.

```
procedure GetReplicaSet
      (* Called remotely by clerks *)
      if not StillMaster() then
            RestartElection();
            raise exception NotMaster;
      end if;
      return Replicas(r);
end GetReplicaSet;

procedure AddReplica(x)
      (* Called remotely by clerks *)
      if not StillMaster() then
            RestartElection();
            raise exception NotMaster;
      end if;
      Stop accepting Write requests;
      Wait for in-progress Writes to complete;
      parfor s ∈ (Active ∪ {x}) do
            try
                  Replicas(s) := Replicas(r) ∪ {x};
            on exception
                  RestartElection();
                  raise exception Failed;
            end try;
      end parfor;
      RestartElection();
end AddReplica;

procedure DeleteReplica(x)
      (* Called remotely by clerks *)
      if not StillMaster() then
            RestartElection();
```

```
        raise exception NotMaster;
    end if;
    Stop accepting Write requests;
    Wait for in-progress Writes to complete;
    parfor s ∈ (Active − {x}) do
        try
            Replicas(s) := Replicas(r) − {x};
        on exception
            RestartElection();
            raise exception Failed;
        end try;
    end parfor;
    if x ∈ Active then
        try
            Replicas(x) := Replicas(r) − {x};
        on exception
            RestartElection();
            raise exception Failed;
        end try;
    end if;
    RestartElection();
end DeleteReplica;


process Election
    ...
    (* Check if best candidate for master *)
    maxService := max{service(s) : s ∈ Con};
    P := {s ∈ Con : data(s) = service(s) and
        prospective(s) ≥ maxService},
        sorted in descending order of goodness;
    if
        No s ∈ Con is enslaved to a replica other than r and
        r = P[1] and
        ( |P| = 1 or r ∈ Replicas(P[2]) )
    then
        myState := PotentialMaster;
    else
```

```
            RestartElection();
            exit process;
        end if;
        ...
    end Election;


    process Recovery
        ...
        (* Advance prospective epoch *)
        ...

        (* Reconcile replica sets *)
        parfor s ∈ Active do
            Replicas(s) := Replicas(r);
        end parfor;

        (* Perform in-progress reconciliation *)
        ...
    end Recovery;
```

2.6 Clerk failover

In this section, we explain how the `Ar` clerk keeps track of which replica is master, and how it fails over from an old master to a new one. This algorithm is essentially independent of the replication algorithm run by the servers, and is much simpler.

To keep track of the master, the clerk runs a background process that takes it through three states: **Searching**, **InTouch**, and **OutOfTouch**. Initially, the clerk is in the **Searching** state. Upon entering this state, the clerk gets a list of replicas that might be master (from the name service, in our implementation), and calls each candidate in turn. If the candidate is master, the clerk enters the **InTouch** state. If the candidate is not master or the clerk cannot contact it, the search goes on to the next candidate on the clerk's list. If the list is exhausted, the clerk pauses for a while to moderate the load on the network and servers, then starts over with a fresh copy of the list (in case the list has changed). If this search goes on for too long, the clerk moves to the **OutOfTouch** state, but it continues looking for the master in the same way.

The clerk's handling of client requests depends on which state it is in. While in the **InTouch** state, the clerk forwards client requests to the master. If a call to the master fails—either the clerk can no longer communicate with the master, or the called replica reports it is no longer master—the clerk moves to the **Searching** state. If this state transition occurs while a call to the master is still in progress, the clerk remembers the incomplete call so it can be retried later. While the clerk is in the **Searching** state, it accepts new requests from clients but causes them to block, hoping it will know the master soon. If the clerk reaches the **OutOfTouch** state, it concludes there is serious trouble, and responds to all client requests by raising the ServiceUnavailable exception (including those requests that are blocked or remembered as well as new ones). If instead the clerk reaches the **InTouch** state again while some requests are still blocked or remembered, it forwards those requests to the new master.

Forwarding saved requests to the new master after a failover raises a problem. The clerk does not know whether a write request that was still in progress when it left the **InTouch** state was committed or not. It is possible that the request never reached the old master or was backed out in the last **Recovery** phase, but it is also possible that the request was completed by the old master or was committed during the last recovery. So when client a requests an operation, a's clerk might forward the operation twice, and if the master blindly performs it twice, the semantics of `Ar` could be violated: There is no guarantee that a client b of some other clerk has not read the result of the operation and overwritten it with a new value between the two forwardings. If b then reads the result again, what it sees is not equivalent to any serialization of the requests a and b submitted to their clerks.

The solution to this problem used in our `Ar` implementation is a byproduct of the caching mechanism described in Section 3 below. In our caching mechanism, cache consistency is guaranteed by requiring a clerk to hold a *write token* whenever it has data in its cache that has been modified but not yet written back to its master. Clerks also hold write tokens while they are waiting for writes to complete, thereby preventing other clerks from writing while a local write is pending. Thus, we prevent the scenario outlined above: while a is uncertain about whether its write has completed, it retains its write token, so b cannot perform a write in the interim. So at worst, a may request the same write twice in succession, with no other writes intervening; this is no problem because writes are idempotent in `Ar`. We expect to use the same technique in our file system implementation.

Other solutions, independent of the caching mechanism, are also possible. For instance, suppose each clerk were to generate a unique identifier for each of its write requests, and these identifiers were stably recorded by the replicas along with the modified data. Then, if a clerk were to submit the same operation twice, the new master would recognize the duplication and report success to the clerk without redoing the operation.

These solutions both depend on the correctness of the clerks, but not in a way that violates our safety conditions for the algorithm. By running these algorithms incorrectly, a Byzantine clerk can at worst make it appear that one of its own clients has requested the same operation twice. But in general, we do not claim to (and cannot) prevent a Byzantine clerk from attempting arbitrary actions on behalf of clients that use it; we can only prevent the clerk from succeeding in doing things that its clients are not authorized to do.

3 Caching Algorithm

Besides replication, the `Ar` implementation models one other important feature of our proposed file system, namely *caching.* To speed its response to client requests, the clerk on each `Ar` client machine keeps a cache of values that were recently read or written by client programs running on the machine.

The clerk manages its cache using a *delayed write-back* policy. That is, a value modified by an `Ar.Set` operation is not immediately written through the cache to the master; instead, it is first written into the cache (making the cache entry *dirty*—different from the master's value), then given to the master later (*cleansing* the cache entry). Any of several events may trigger cache cleansing. The cache may be filling up with dirty entries, some of which need to be cleansed and displaced to make room for more recently used values. A client may have issued an `Ar.Sync` request that requires the clerk to write one or more dirty values back to the service's stable storage. The clerk may have a policy that an entry must be cleansed within (say) five minutes of being dirtied, to minimize the amount of work that a client may lose when its machine crashes.[17] Or the clerk may be asked to cleanse an entry by the cache consistency protocol. We discuss this protocol next.

[17] In a file system, such long write-back delays help to reduce the load on the servers, because files are often overwritten or deleted within the delay period. The current `Ar` implementation actually does not include a timer to cause dirty entries to be written back eventually, nor does it ever displace a cache entry due to lack of room, but these features

3.1 Cache consistency

Our `Ar` implementation uses *tokens* to preserve cache consistency. A token for an element of `Ar` is a permit to hold a cached copy of that element. A read token gives permission to hold a clean copy; a write token gives permission to hold a dirty copy. Because every client operation goes through its clerk's cache, tokens are quite important to the operation of `Ar`. Every `Ar.Set` requires either obtaining or already having a write token, and every `Ar.Get` requires either obtaining or already having a read (or write) token.

Tokens are issued to clerks by the master, and the master preserves cache consistency by maintaining the invariant that for each `Ar` element, either no clerks hold tokens, one clerk holds a write token, or one or more clerks hold read tokens. Read and write tokens are never held simultaneously. Under this invariant, there is precisely one current version of an element's value at each moment—if a write token for element `i` is outstanding, the version in the token holder's cache is current; otherwise the master's version is current. An `Ar.Get(i)` operation always returns the current value of element `i`. The clerk's interface to the master contains three operations that deal in tokens: `ArMaster.Get` gets a value and a read token for it from the master, `ArMaster.Set` gets a write token and sets a value, and `ArMaster.Cleanse` sets a value for which the clerk already has a write token, without ceding the token.

When a clerk requests a token that would conflict with one or more outstanding tokens, the master calls back to their holders, ordering them to cede the conflicting tokens. A clerk that is told to cede a token does so immediately (after cleansing the cache entry it covered, if it was dirty), without waiting to acquire any other tokens. There are no operations that require more than one token. Thus the token protocol is not subject to deadlock.[18]

Although this protocol is basically simple, it becomes a bit more complex when one considers unusual situations that can arise due to reordering of messages on the network. In particular, while a clerk has a token request pending, the master may call the clerk back and ask it to cede a token on the same object. It is difficult for the clerk to determine whether the master has

could easily be added.

[18] It appears that Echo will have operations that require more than one token—for example, moving a directory to be under a different parent—so the deadlock problem will not be so simple there. We currently plan to avoid deadlocks in Echo by requiring clerks doing multi-token operations to acquire the tokens in a fixed order.

already granted the token and is asking for it back, or is asking for a token that the clerk already (perhaps unknowingly) held. We spent quite a bit of effort trying to develop a neat, efficient solution to this problem—without success—and at last fell back on the simple technique used in CMU's Andrew system [14]. When a request and a callback cross as just described, the clerk responds positively to the callback and internally marks the request as aborted. When the aborted request returns, the clerk retries it. Such retries occur rarely, so we consider this solution adequate.

3.2 Token recovery

The token mechanism adds one step to the recovery phase of our election algorithm. Just before going into service, a newly elected master must perform *token recovery.* Token recovery consists of contacting every clerk to find out what tokens it holds. This step is necessary because in our caching implementation, a token is a record kept only in the volatile memory of the master and of the clerk that holds it. Therefore when a master crashes, the only remaining record of what tokens are outstanding is contained in the memory of the clerks, and the next master to be elected must contact them to reconstruct the lost state.

A newly elected master running token recovery knows which clerks to contact by looking at its *clerk list.* A clerk list is an up-to-date record listing all clerks that hold one or more tokens. The clerk lists are kept only in the primary memory of the replicas, and not on their stable storage. During normal operation of the service, the master informs its slaves whenever a new clerk requests a token and whenever an old clerk gives up all its tokens; the slaves are informed before the call returns to the clerk. When the master crashes, the former slave that is next elected master asks each clerk on its clerk list what tokens it holds, and aggregates this information to construct a new list of outstanding tokens. We modify the *goodness* function of the replication algorithm so that replicas with non-empty clerk lists are preferred in the election—a replica with a non-empty clerk list always has a higher *goodness* value than one with an empty list. Replicas with non-empty clerk lists must be preferred because the clerk lists are not recorded on stable storage: two replicas can have equally recent epochs with only one of them having a clerk list because the other replica crashed and restarted, losing its primary memory. Note that because the addition of a new clerk changes the clerk list on the master and all its slaves before returning to the clerk, all non-empty clerk lists on replicas with current epochs are equally good—any differences are only over clerks that hold no tokens.

Because clerks are not trusted, the new master must be prepared for the information it gets back from the clerks to be incorrect. Therefore, whenever a clerk claims to hold a token, the master must check that this token does not conflict with tokens that are already on its list of outstanding tokens; if the token does conflict, it is not granted. Moreover, in the real file system, the master will need to check that the clerk has the right to hold the tokens it claims—that is, it will have to do the same user authentication and permission checking it does when tokens are requested during normal operation. Even with these checks in place, it is possible for a Byzantine clerk to acquire tokens during recovery that it did not originally have, perhaps even taking them away from correct clerks that held them legitimately. But this can happen only if the Byzantine clerk is able to authenticate itself as a user who is authorized to have those tokens, so security is not violated.

Finally, since the clerk lists themselves are not kept on stable storage, token state is lost entirely if all replicas crash at once (or very closely together). We consider this behavior acceptable in such a rare event. (Practically speaking, the only thing likely to make all the replicas crash at once is a massive power failure, which is likely to take out the client machines as well.)

We have chosen this design because it optimizes the common case, token acquisition during normal operation, at the expense of the uncommon case of crash recovery. The master can give out tokens quickly because it does not have to write them to stable storage or tell its slaves about them; it has to tell the slaves only when a new clerk begins using the service or an old one stops, which are much less frequent events. It is an open question whether the cost of token recovery is too high. If token recovery in Echo causes too long a delay in resuming service after a master crash, we may be forced to use token replication instead—that is, the entire token list would be replicated in each replica's volatile memory, just as the clerk list is replicated in the scheme we have just described.

3.3 Clerk failover

Section 2.6 above describes how clerks fail over from an old master to a new one.

A clerk does not discard its cache upon failover unless its new master tells it that the cache is no longer valid—otherwise it always proceeds with the hope that the new master will recover the token database and preserve the cache's validity. Even when a clerk has been unable to contact its master for so long that it has entered the **OutOfTouch** state and has begun raising `ServiceUnavailable` to all client requests, it still holds on to its cache until it gets back in touch.

3.4 Token timeouts

Clerk failures cause a problem for the caching system because a failed clerk may have been holding some tokens that are later needed by other clerks. It is not safe for the master simply to cancel a clerk's token and give out a conflicting one if the clerk does not respond to a callback, because the clerk may not actually be down—it may only be partitioned from the master by a network failure. If the master gives out a conflicting token, cache coherence is violated, and this violation will become apparent as soon as the partition is repaired.[19]

`Ar` uses a strict form of *token timeout* to allow tokens to be recovered from failed clerks with no danger of cache incoherence. When the master issues a token to a clerk, the token is good only for a limited time, called its *timeout*. Until the timeout expires, the master promises not to issue any conflicting tokens without first calling the clerk back and getting it to cede the token. If the master is unable to call the clerk back, it must wait for the timeout to expire before it can cancel the token. Thus the clerk knows that its token has not been taken away if it has not yet timed out.[20]

At any time, a clerk can ask the master to *refresh* a token. If the master agrees, it restarts the timeout period. A clerk can ask the master to refresh a token that has already timed out, but the master will not do so if it has already canceled the token and issued a conflicting one.

This token timeout scheme imposes three kinds of costs on the system:

- **C1.** Each clerk must occasionally call the master to refresh its tokens.
- **C2.** If clerk p needs a token that conflicts with one held by clerk q, and clerk q has recently gone down (or become partitioned away from the master), clerk p must wait, in the worst case for the full timeout period.

[19] Some distributed file systems do not attempt to prevent coherence violations of this sort. For example, CMU's Andrew system [14] permits them to arise, but limits their duration to ten minutes.

[20] Note that we are using the assumption that the master and clerk have clocks that run at approximately the same rate (within a known factor ρ that can be taken into account in the timeout test).

- **C3.** When a newly elected master is performing token recovery, it may find itself unable to contact some of the clerks on its clerk list. To guarantee cache coherence, the new master must be conservative and assume the worst about what tokens the unreachable clerks hold, until it is sure they have all timed out. In the interim, some or all requests from other clerks have to wait, in the worst case for the full token timeout period.

One can trade off costs C2 and C3 against cost C1, because a longer token timeout lowers C1 while raising C2 and C3, while a shorter one does the opposite. There are also many techniques and policy decisions about when to refresh that can improve things. Our current belief is that the following techniques will work well in Echo.

First, to keep the cost of refresh messages (C1) manageable, all tokens for a given service held by a given clerk are associated with a *session* between the clerk and service, and the session is what times out and needs to be refreshed, not the individual tokens. Thus a service receives at most one refresh request per clerk per timeout period.[21]

We then reduce C1 a bit more by declaring that all successful calls from a clerk to the master also refresh the calling clerk's session, a technique we call *implicit refresh.* Thus a session that is actively in use (satisfying cache misses and write-back requests) is kept refreshed with no additional messages.

Finally, we reduce C1 still more using *lazy refresh.* Under this approach, clerks do not try to keep their sessions fresh at all times; instead, they refresh them only as needed. Whenever a clerk needs to use a cached object, it checks whether the session is expired, about to expire, or not due to expire soon. If the session is expired, the clerk attempts to refresh it before using the cached object; if the refresh succeeds, all is well; if not, the clerk creates a new session and discards the cache contents from the old one. (We assume the clerk's write-back timeout is shorter than the session timeout, so the clerk will never let a session expire without first trying to write back all its dirty data.) If the session is about to expire, the clerk refreshes it to stave off expiration, in parallel with using the cached object. If the session is not due to expire soon, the clerk does nothing about refresh. The master cooperates with this approach by not revoking a timed-out session immediately when

[21] To be precise, slightly more than one, because sessions are refreshed somewhat before they time out to avoid problems with network delays and clock skew.

some other clerk needs one of its tokens; instead, it tries to call back the session owner and revokes the session only if there is no response.

Under this approach, a clerk never does extra calls to the master to refresh its session unless it is running purely out of its cache for a long time, with no misses and no write-backs. If the clerk has data sitting in its cache, but clients are no longer using it, the clerk does not refresh it. (We call such a session *quiescent*.) If a clerk crashes while its session is quiescent, the cost of reclaiming its tokens in cases C2 and C3 is only one remote procedure call timeout, not a whole token timeout.

There are a few drawbacks to the lazy refresh approach as well. The first token to be reclaimed from a failed clerk always costs an RPC timeout, even if the clerk has been down for a long time. Also, clients have to wait for a session refresh on their first operation when they stop being quiescent, even if the operation hits in the cache. Managing the slave clerk lists costs a bit more, too. Slaves should have quiescent clerks on their lists, and should know that they are quiescent. If the slaves did not list quiescent clerks at all, then a change of mastership would cause those clerks to lose their tokens. And if the slaves did not know which listed clerks were quiescent, then whenever a quiescent clerk was down during a mastership change (C3), the full token timeout would be incurred to get back its tokens, not just an RPC timeout. Therefore, the master must send its slaves a message whenever a clerk's session timeout expires, then another when the expired session is either refreshed or revoked.

We plan to study and refine these mechanisms further before Echo is complete.

3.5 Unstable updates

Because our caches use a write-back policy, even after a client's call to `Ar.Set` returns successfully, the client cannot be sure the update will not be lost, unless it is willing to pay the cost of an `Ar.Sync`. Obviously, updates can be lost if the client machine fails before its clerk has finished writing them to the service. But updates can also be lost if a client machine is partitioned from the master for so long that it *appears* to have crashed—so long, that is, that the master cannot contact it when it needs to do a callback. In that case, as we described in the last section, the master revokes the clerk's session, and the clerk must discard its cache when it reconnects with the master and discovers the problem. We call data *unstable* when it has been updated by a call to `Ar.Set`, but the update is still susceptible to being lost;

that is, when it is not yet recorded in the stable storage of the master and all slaves.

In `Ar`, the token mechanism guarantees that unstable data can be read only by clients on the same machine as the client that wrote the data. This is true because `ArMaster.Set` and `ArMaster.Cleanse` are synchronous to stable storage, and the clerk refuses to give up its write token until they return. We believe this is a useful and sensible guarantee because machines are units of failure—when a machine crashes, all the programs running on it fail, but those on other machines keep going. The adverse impact of lost updates on the system as a whole is lessened if the machines that remain up are known not to have seen any updates that were lost in the crash.

One cost of this guarantee is that if two programs on separate machines are communicating through the service, they cannot exchange data any faster than the service can write it to stable storage. For this reason, we have contemplated removing the guarantee in Echo, but doing so without weakening other safety guarantees is rather tricky, and beyond the scope of this paper.

4 Extensions

In this section, we describe two useful extensions to the basic replication scheme we have discussed so far.

4.1 Reading from slaves

In our `Ar` implementation, all update requests and all read requests are directed to the master replica. It is easy to extend the algorithm so that slaves are also able to satisfy read requests.

The token mechanism is used to ensure consistency. When reading a file (or directory), the clerk first acquires a read token from the master, then sends its read requests to any replica, either the master or a slave. Because the clerk has a read token, there cannot be an update in progress on this file, so each slave's copy is identical to the master's, and the clerk gets the same result no matter which it reads.

Although the master is still responsible for granting read tokens, this scheme allows slaves to handle the load of providing file data—the bulk of the read load for most files longer than a few network packets. The scheme can thus be used for load balancing between master and slaves—even though the master does somewhat more work than the slaves when managing tokens and handling write requests, the slaves can be given a larger share of the read load to compensate. It seems attractive for clerks to statically determine which active replica to use for reading a file by hashing the file's identifier (which is the same for all clerks). Thus each replica handles reads for a different, disjoint set of files, improving the locality of reference in the replicas' caches. A bias towards reading from slaves rather than the master can be built into the hash function.

In this scheme, nothing prevents a Byzantine clerk from reading a file without having the read token, and if it does so, it may be given incorrect answers by the server. However, correct clerks ensure themselves of getting correct answers by acquiring read tokens as they are supposed to. Further, the master ensures that a Byzantine clerk cannot give up its write token while it still has a write in progress (which would cause problems for correct clerks), by being careful not to ask a clerk to give up its write token until any writes that the master is currently performing for the clerk have completed.

4.2 Ensuring that all data is replicated

Our replication algorithm is willing to provide service even when only one replica is up-to-date and writing new updates to stable storage. The remaining replicas making up the majority can be witness replicas or can be non-witness replicas that are engaged in catchup reconciliation. This makes our algorithm vulnerable to the following problem: If the single up-to-date replica crashes, the system must wait for it to come up again before providing service. If it never comes up, the system can never again provide service (without manual intervention). Because the crashed replica is the only one that has all the updates, no other replica is eligible to become master.

For example, consider a configuration with two non-witness replicas r and s, and one witness replica t. Suppose that initially all three are up and communicating, and that both r and s have data epoch equal to service epoch. Then r crashes. An election is held, and s is elected master. The system enters service and client programs perform additional updates; then s crashes. Now replica r comes up; however, since r is missing some updates that s has, r is not eligible to be master. (The epochs held by the witness replica t enable the election algorithm to detect that r is not eligible.)

For configurations with only two non-witness replicas, the problem is unavoidable. We must either be willing to provide service when one of the non-witness replicas is down, or accept availability that is less than that of a non-replicated system.

With more than two non-witness replicas, we can do better. We can modify the algorithm to insist that before entering service, at least k replicas must be up-to-date, where k is a parameter that can be set by the system administrator. We change the recovery algorithm so that it blocks after beginning catchup reconciliation and before advancing the service epochs, until enough replicas have caught up to make a total of k up-to-date.

The choice of k involves a tradeoff. Setting $k > 1$ increases recovery time, but guarantees that all updates are replicated. The system is unavailable during this increased recovery time, but the replication improves its availability once it does go into service (because more replicas have up-to-date data and are therefore eligible to be master) and also makes the data more likely to survive disk crashes. As an additional (minor) benefit, catchup reconciliation is likely to be completed faster if it is performed before the system goes into service, rather than in background while the system is in service, because in the latter case it has to compete for processing time against the requests generated by clients. On the other hand, if the system can nearly always complete catchup reconciliation in background before the newly elected master crashes, it makes sense to set $k = 1$ to get better availability through faster recovery.

Note that not all possible combinations of choices for k and the number of witnesses w in a system configuration make sense. Of course, it is necessary that $0 < k \leq |Replicas|$ and $0 \leq w < |Replicas|$. Also, there is little point in choosing more than $\lceil (|Replicas|+1)/2 \rceil - k$ witnesses. Having more witnesses only permits the system to form majorities that include fewer than k non-witnesses and thus cannot go into service.[22] For instance, if there are four non-witnesses and $k = 1$, then there should be at most three witnesses, while if there are four non-witnesses and $k = 2$, then there should be at most one witness. Also, if k is chosen larger than $\lceil (|Replicas| + 1)/2 \rceil$, the number of replicas that must be up for the system to go into service becomes more than a majority, but such a choice may still be sensible for some applications.

Parenthetically, it may seem that a system configured with $k > 1$ could safely operate in a read-only mode whenever a majority of replicas are up and connected but only some number of them $k' < k$ are up-to-date; however, this idea does not work. When the system last crashed, some update operations were in progress, and in the new read-only service period, clients must be able to read the values that these operations were attempting to

[22] However, such a majority can elect a master and perform catchup reconciliation on those non-witnesses that are up.

update. Therefore, before the system can go into service, it must decide which of these updates will be carried out and which ignored. Only the replicas that are active in the current read-only service period know the outcome of these decisions. Any inactive replica that was up-to-date at the time of the last crash must now be considered to be out-of-date, because it does not know the outcome of the decisions: if an inactive replica were later allowed to become master without being reconciled with the presently active replicas, it could make different decisions, causing the system to give inconsistent semantics to its clients. Thus, even to enter read-only service, the system must mark inactive replicas as out-of-date (say, by advancing the service epoch on the active replicas, as in our recovery algorithm). Since only the active replicas are up-to-date, we have only k' up-to-date replicas, which violates our original requirement of always having $k > k'$ up-to-date replicas.

5 Summary and Status

We have presented an algorithm for data replication. The algorithm permits the set of replicas to change dynamically, allows the use of witness replicas, and supports consistent caching of data by client machines. During normal service, an update requires only a single remote procedure call from a client to the master, plus a remote procedure call from the master to each slave. After a crash, the outcome of each update that was in progress is decided by a recovery phase that executes before the next service period. The algorithm does not depend on distributed atomic transactions.

Currently, we are exploiting the techniques described in this paper in designing and implementing the Echo file system. In our work on `Ar`, we assumed that a server CPU and a disk came bundled together in a single unit, called a replica. In Echo, our system structure is more general and more flexible. We support the use of multi-ported disks, that is, disks that are directly accessible to several CPUs, each of which might fail independently. Server CPUs and disks can be replicated (or not) independently of each other. The `Ar` problem of finding up-to-date replicas reappears in Echo as the problem of finding up-to-date disks. As in our work on `Ar`, we are employing monotonically increasing epoch variables and majority election to solve this problem. We will report more fully on Echo in future papers.

References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[2] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[3] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. A simple and efficient implementation for small databases. Research Report 24, DEC Systems Research Center, January 1988.

[4] Michael Burrows. *Efficient Data Sharing.* PhD thesis, Churchill College, University of Cambridge, September 1988.

[5] Jo-Mei Chang. Simplifying distributed database systems design by using a broadcast network. In *SIGMOD '84 Proceedings*, pages 223–233. ACM SIGMOD, June 1984.

[6] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[7] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[8] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[9] Hector Garcia-Molina, Frank Pittelli, and Susan Davidson. Applications of Byzantine agreement in database systems. *ACM Transactions on Database Systems*, 11(1):27–47, March 1986.

[10] David K. Gifford. Weighted voting for replicated data. In *Proc. 7th Symp. on Operating Systems Principles*, pages 150–159. ACM SIGOPS, December 1979.

[11] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. 11th Symp. on Operating Systems Principles*, pages 155–162. ACM SIGOPS, November 1987.

[12] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[13] Sushil Jajodia and David Mutchler. Dynamic voting. In *Proc. ACM SIGMOD 1987 Annual Conference*, pages 227–238. ACM SIGMOD, May 1987.

[14] Michael L. Kazar. Synchronization and caching issues in the Andrew file system. In *Winter Conference Proceedings*, pages 27–36. USENIX Association, February 1988.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.

[16] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 6(2):254–280, April 1984.

[17] Leslie Lamport. The part-time parliament. Research Report, DEC Systems Research Center, to appear, April 1989.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Computer Systems*, 4(3):382–401, July 1982.

[19] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–28, January 1984.

[20] Butler W. Lampson. Designing a global name service. In *Proc. 5th Symp. on Principles of Distributed Computing*, pages 1–10. ACM SIGACT-SIGOPS, August 1986.

[21] Keith Marzullo and Frank Schmuck. Supplying high availability with a standard network file system. Technical Report 87-888, Dept. of Computer Science, Cornell University, December 1987.

[22] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[23] Jehan-Francois Pâris. Voting with witnesses: A consistency scheme for replicated files. In *Proc. 6th International Conference on Distributed Computer Systems*, pages 606–612. IEEE Computer Society, 1986.

[24] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A network transparent, high reliability distributed system. In *Proc. 8th Symp. on Operating Systems Principles*, pages 169–177. ACM SIGOPS, December 1981.

[25] Paul Rovner, Roy Levin, and John Wick. On extending Modula-2 for building large, integrated systems. Research Report 3, DEC Systems Research Center, January 1985.

[26] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[27] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proc. 9th Symp. on Operating Systems Principles*, pages 49–70. ACM SIGOPS, October 1983.