CITI Technical Report 04-01

Replication Control in Distributed File Systems

Jiaying Zhang
jiayingz@eecs.umich.edu

Peter Honeyman
honey@citi.umich.edu

ABSTRACT

We present a replication control protocol for distributed file systems that can guarantee strict consistency or sequential consistency while imposing no performance overhead for normal reads. The protocol uses a primary-copy scheme with server redirection when concurrent writes occur. It tolerates any number of component omission and performance failures, even when these lead to network partition. Failure detection and recovery are driven by client accesses. No heartbeat messages or expensive group communication services are required. We have implemented the protocol in NFSv4, the emerging Internet standard for distributed filing.

April 1, 2004

Center for Information Technology Integration
University of Michigan
535 W. William St., Suite 3100
Ann Arbor, MI 48103-4978

Replication Control in Distributed File Systems

Jiaying Zhang and Peter Honeyman
jiayingz@eecs.umich.edu honey@citi.umich.edu

1. Introduction

In modern distributed systems, replication receives particular attention for improving performance and availability: failure can be hidden from users and applications if they can obtain data services from an identical replica; replication can improve performance by scaling the number of replicas with demand and by offering nearby copies to services distributed over a wide area.

A fundamental challenge with replication is to maintain data consistency among replicas. In a replication system, the value of each logical item is stored in one or more physical data items, referred to as its *copies*. Each read or write operation on a logical data item must be mapped to corresponding operations on physical copies. Exactly when and how these mappings are carried out determines the consistency guarantees provided by the system and the cost of replication.

An ideal distributed file system provides applications *strict consistency*, i.e., a guarantee that all I/O operations yield identical results at all nodes at all times [2]. To enhance data availability, a replication system must tolerate common component failures. To reduce overhead due to replication, a replication control protocol should minimize the number of physical accesses required to implement a logical access. In practice, these three goals always conflict with each other and the trade-offs among them need to be considered.

In this paper, we present a replication control protocol for distributed file systems that tolerates a large class of failures, guarantees strict consistency, and imposes little overhead on normal reads. We observe that not all applications need strict consistency - Often, ordered writes suffice, i.e., although applications do not necessarily see updates simultaneously, they are guaranteed to see them in the same order. Therefore, our system also provides the support for this consistency model in case of failures; as we shall see, it allows applications to trade consistency for availability. In the following discussion, we follow Lamport [17] and refer to ordered writes as *sequential consistency*.

There are many examples of replication schemes in distributed file and database systems. The work described here is novel in the following ways.

First, our protocol uses a primary-copy with server redirection scheme that offers strict or sequential consistency without imposing any overhead on normal reads.

Second, our design takes into account file system workload characteristics to improve replication performance. Distributed file systems typically encounter workloads in which write sharing is rare. Furthermore, when a file is opened for writing, there is usually a burst of updates. Unlike the traditional primary-copy [6] or two-phase locking [13] approaches, our system dynamically binds a primary server when a client opens a file for writing. This offers two benefits: first, it provides superior read performance by allowing a client to access data from a nearby replication server if the referred file is not under modification. Second, when a client modifies a file, only file open and close operations require additional concurrency control messages for replication support.

Third, most recoverable systems detect failures with periodic heartbeat or probing messages, inducing cost to normal operations. In our system, failure detection and recovery are driven by client access requests. No heartbeat messages or expensive group communication services are required.

Fourth, we have implemented the system as an extension to a standard Internet filing protocol, NFSv4 [9], which promise widespread acceptance and deployment.

In the remainder of this paper, we describe the types of failure that we anticipate in a distributed system. Then we introduce our replication control protocol and the recovery procedures in the case of failures. We use a result from database theory to prove that our protocol can guarantee sequential consistency. After that, we

describe ways to support strict consistency and discuss the related work.

2. Failure Models

A system *fails* if it does not adequately provide the services for which it is designed. There is a well-studied hierarchy of failures in distributed systems, *omission failure*, *performance failure*, and *Byzantine failure* [9].

Omission failure occurs when a component fails to respond to a service request. Typical omission failures include server crash or communication link outage. *Performance failure* occurs when a system component fails to respond to a service request within the time limit specified for the delivery of that service. Occasional message delays caused by overloaded servers or network congestion are examples of performance faults. In *Byzantine failure*, components act in arbitrary, even malicious ways. Compromised security can lead to Byzantine failure.

An important subclass of omission and performance failures is network partition. A partition is a collection of connected servers and clients isolated from the rest of the system.

Although security breach is increasingly common in the Internet, Byzantine failure is beyond the scope of our work. By implication, we rely on the distributed file system to provide authorized communication. This narrows our focus to two kinds of failure: crashed nodes and partitioned networks. Furthermore, we assume that even these failures are rare, although this does not affect the security or correctness of our protocol. Rather, our goal is to develop a system that performs well in the face of typical Internet conditions and application requirements. In the next section, we present the design of the replication control protocol.

3. Protocol Design

In a distributed file system, some nodes are *servers* and some are *clients*. Clients send messages to servers to request service; servers accept the messages, carry out the requests, and return responses to the client. In this paper, we focus on the replication control protocol that guarantees consistency in the face of node and network failure. The mechanisms for locating and managing replicas, as well as implementation and performance details, can be found in a companion paper [7].

Our goal is to provide strict or sequential consistency at little cost to exclusive or shared reads. To realize this goal, we use a primary copy method with server redirection when concurrent writes occur. The strategy differs from the usual primary copy scheme in that it allows late and dynamic binding of the primary server, chosen at the granularity of a single file or directory. We present details in the following subsections.

3.1 File Updates

When a client opens a file for writing, the server it selects temporarily becomes the primary for that file by instructing all other replication servers to redirect further client accesses for that file to it. When the file is closed, the primary server withdraws from its role by re-enabling replication on the other replication servers.

Two or more servers may try to become the primary for a file at the same time. When these servers are in the same partition, contention is always apparent to the conflicting servers. We resolve the conflict by having conflicting servers cooperate: the server that has disabled more replicas is allowed to continue; the server that has disabled fewer replicas quits the process; when a tie happens, the server with bigger IP address is allowed to proceed. If the conflicting servers are in different partitions, and neither of them can collect the acknowledgements from a majority of the replicas, no server can become the primary. We discuss this case further in Section 4.5.

While replication is disabled, the primary server is responsible for distributing updates to other replication servers. Updates must be delivered in order, either by including a serial number with the update or through a reliable transport protocol such as TCP. In addition to distributing the file data written by clients, each update message from the primary server to other replication servers also includes the metadata related to the update, such as the modification time. Every replication server stores the metadata accordingly after updating the file data. As we show in Section 4.4, the stored metadata help to identify the most recent copy of the file during failure recovery. File modification with two replication servers is illustrated in Figure 1.

3.2 Directory Updates

Directory modifications include creation, deletion, and modification of entries in a directory. Unlike file writes, little time elapses between the start and the finish of a directory update, which reduces the likelihood of concurrent accesses to the directory while it is being updated. So instead of redirecting access requests for the directory while an update is in progress, replication servers simply block these accesses until the primary server distributes the update and re-enables replication. Directory modification with two replication servers is illustrated in Figure 2.

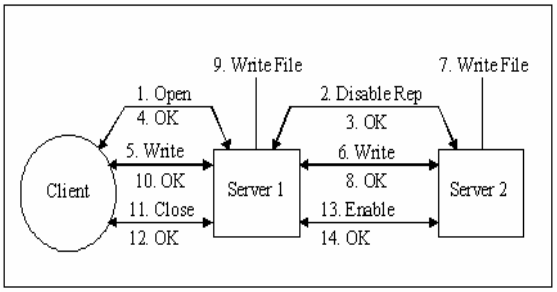

Figure 1: File modification.
1. A client issues an open request to a server. 2. The server instructs other replication servers to redirect requests, making it the primary server for the file. 3. Replication servers acknowledge the request. 4. The primary server acknowledges the open request. 5. The client sends writes to the primary server. 6. The primary server distributes the writes to other replicas. 7. Other servers update the written file. 8. Other servers acknowledge the update. 9. The primary server updates the written file. 10. The primary server acknowledges the client write request. (Steps 5 through 10 may be repeated.) 11. The client issues a close request. 12. The close request is acknowledged. 13. The primary server instructs the redirected servers to re-enable replication. 14. The redirected servers disable redirection and acknowledge the request to re-enable replication.

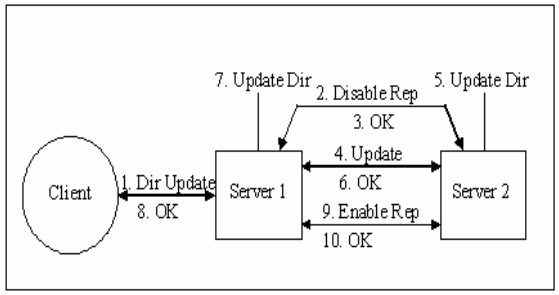

Figure 2: Directory modification.
1. A client issues a directory update request to a server. 2. The server instructs other replication servers to block any access to this directory. 3. Replication servers acknowledge the request. 4. The primary server distributes the update request. 5. Other servers update the directory. 6. Other servers acknowledge the update. 7. The primary server processes the directory update request. 8. The directory update request is acknowledged. 9. The primary server instructs the other servers to re-enable access. 10. The redirected servers restore access to the directory and acknowledge the request to re-enable replication.

4. Failure Recovery

The principle challenge to data replication is maintaining consistency in the face of failure. In this section, we enumerate the failures that can occur and the actions to take to maintain consistency.[1] Using the view change properties and rules proposed by El-Abbadi et al. [1], we are able to prove that our protocol guarantees sequential consistency. The proof and the pseudo-code are given in the next section.

In our design, we assume an asynchronous communication network: there is no bound on the message transmission delays between nodes. In such a network, it is impossible for a node **p** to distinguish between a failure of node **q** (crash failure) or a failure of the communication network connecting **p** and **q** (partition failure). Yet these two kinds of failure can have different effects on file system states: while a failed node can not perform any operations, in a partitioned network, if no control mechanisms are used, nodes in different partitions may operate on the same logical object unwittingly, which leads to inconsistent system states.

[1] Failure recovery for directories is not discussed in detail as it can be viewed as a special case of file write.

To prevent data inconsistency, our protocol maintains an active view among replication servers and allows updates only among the servers contained in the active view. We refer to the server group covered by the active view as *active group*. To ensure the uniqueness of the active group, we follow Cristian and Mishra [8] and define it to be one that contains a majority of the replication servers. In the remainder of this section, we consider the different types of failure and explain the failure recovery procedure for each case.

4.1 Client Crash

Following the specification of NFSv4, each open file is associated with a lease, subject to renewal by the client. In the event of client failure, the primary server receives no further renewals, so the lease expires. Once the primary server decides that the client fails, it closes each file opened on behalf of the failed client. If the client is the only writer for a file, the primary server re-enables replication for the file. Unsurprisingly, the file content reflects all writes acknowledged by the primary server prior to the failure.

4.2 Network Partition

To detect partition failures, every server keeps a table that records the liveness of other replication servers from its point of view. The set of live servers is called the *active view*.

Network partition changes the primary server's active view. The partition is detected when the primary

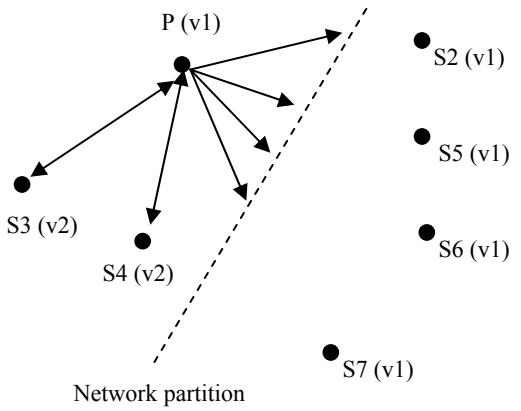

Figure 3.1

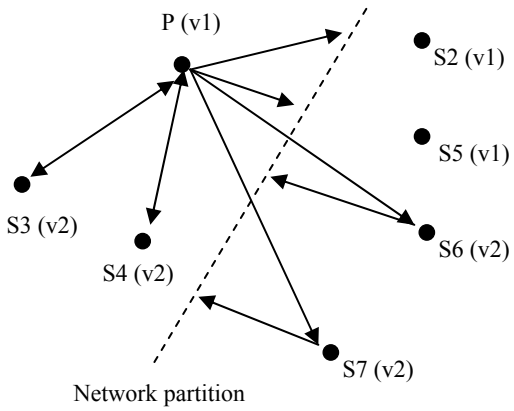

Figure 3.2

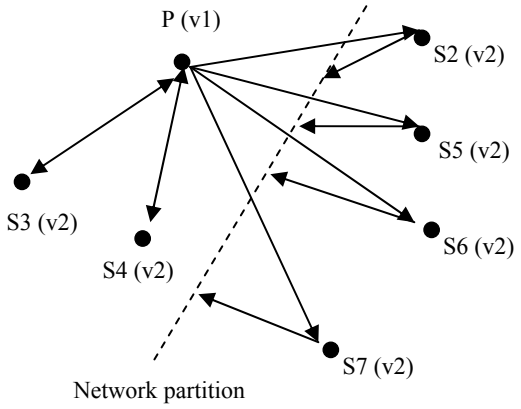

Figure 3.3

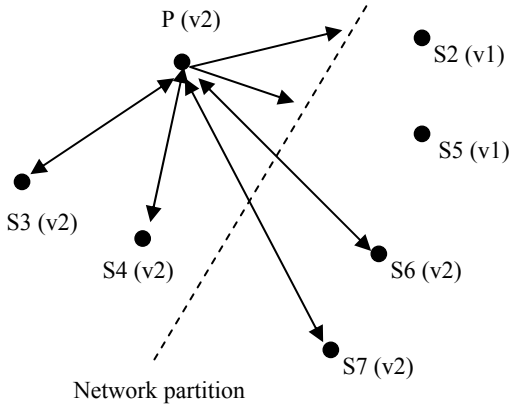

Figure 3.4

Figure 3: Possible situations after the primary server is partitioned in a minority partition.
This figure shows different situations that may occur after the primary server P is separated in a minority partition during processing a client's write request. S2 - S7 represent other replication servers. The value in parentheses denotes the data version of the file copy at the corresponding replication server.

server disables replication, when it distributes updates to other replication servers, or when it re-enables replication. We consider each case in turn.

4.2.1 Detect Partition while Disabling Replication

The primary server constructs its active view for the file as it receives acknowledgements for disabling replication requests. As soon as the active view constitutes a majority, the primary server acknowledges the client open request. A replication server cannot become the primary if it fails to collect acknowledgements from a majority of the replication servers. However, the client should impose a timeout on its open request to avoid waiting forever.

4.2.2 Detect Partition during Update Distribution

The primary server forwards each client write request to the replication servers in its active view. The primary server updates its local copy and acknowledges the client request after it receives update acknowledgements from a majority of the replication servers. Any replica that fails to acknowledge an update is removed from the active view.

If the active view shrinks to less than a majority during processing a client write request, different situations may occur, as illustrated in Figure 3.1-3.3. Because the primary server can not determine the data version in the majority partition, it fails the client write request. Some replication servers may have applied the update; consequently the file content is inconsistent among replication servers. However, because replication is still disabled at these servers, and as we show in Section 4.4, our failure recovery procedure guarantees that the system converges in the majority partition be-

fore re-enabling the replication, inconsistent data accesses are prevented.

Figure 3.4 shows a special situation, in which network failure places the primary server in a minority partition after it assembles a majority view for the current update. As described previously, the primary server updates its own copy and acknowledges the write. At this time, there must be at least one fresh copy of the file in the majority partition. In Section 4.4, we describe how that fresh copy is found and used to update other replication servers to assure data consistency after failure recovery.

4.2.3 Detect Partition while Re-enabling Replication

Partition that is detected when the primary server processes a client close request does not affect data consistency, as all replication servers in the active view should have the same file contents. Therefore, the primary server acknowledges a client close request immediately. At the same time, it sends its active view to other replication servers in the view, and re-enables the replication on them. Any server outside the active view may have stale data, so the servers that are re-enabled replication refuse any later requests coming from a server not in the active view. A failed replica can be re-added to the active view only after it synchronizes with the up-to-date copy.

4.3 Replication Server Crash

Replication server crash can be regarded as a special case of network partition if we consider the crashed server as a single partition. Therefore the same handling procedure applies. When the primary server detects a replication server failure during disabling replication or distributing updates, it removes the failed server from its active view. At the time of re-enabling replication, the primary server sends its active view to other live replication servers. These servers refuse any latter requests from a server outside the active view. A failed replica has to synchronize itself with a live replication server before being re-added into the active view.

4.4 Primary Server Failure Recovery

A client detects primary server failure (crash failure or network partition that separates the primary server in a minority partition) when an access request times out. In this case, the client selects a live replication server and informs it of the failure. After verifying the primary server failure, the selected replacement works with other replication servers to become the new primary, as follows.

The replacement first asks all other replication servers for permission to become the new primary and the modification time of the file copy at each replica. The failure may have occurred during replication disabling stage; consequently some replication servers may not have received the replication disabling requests from the failed primary. In this case, these replicas simply grant the replacement the authority to become the primary. If the request comes to a replica that has been disabled replication, that server verifies the primary server failure, switches the primary, and acknowledges the request. We note that upon receiving a recovery request, a replication server should first check the state of the original primary server to avoid switching the primary blindly under unusual network conditions; e.g. network connections may not be transitive.

The replacement becomes the new primary if it receives the acknowledgements from a majority of the replication servers. It then determines which replicas have stale data by comparing the received modification times and synchronizes these servers with the freshest file copy, which it may first have to retrieve for itself. Following that, the replacement primary constructs a new active view, distributes it to other replication servers in the view, and re-enables their replication.

The replacement responds to the client after finishing the recovery. The client then engages in the client-side recovery mechanism for server migration, part of the NFSv4 standard. In brief, this entails cleaning up state associated with the old server, reopening the file on a new server, and reissuing the failed request. If the failed I/O is a write, it is indeterminate whether the state of the recovered file reflects that write operation. However, this does not compromise data consistency as repeating a write request is safe in NFSv4.

Other clients may also detect the primary server failure and initiate server recovery on different replication servers. If the recovery is not complete, the conflict is resolved through the same procedure as mentioned in Section 3.1. When replication is later re-enabled, all such clients receive the acknowledgements and engage in client-side recovery. If a client detects the failure after the recovery is complete, it receives an immediate response and starts client-side recovery.

4.5 No Majority Partition

As described above, the failure recovery procedure starts when the primary server is isolated in a minority partition. However, if there are multiple partitions and no partition includes a majority of the replication servers, a new primary can not be elected until the partitions heal sufficiently to allow a quorum to assemble. In this case, read requests can be satisfied, but any write requests are refused.

5. Proof of Correctness

In databases, *one-copy serializability* requires that the concurrent execution of transactions on replicated data be equivalent to a serial execution on non-replicated data [4]. In replicated file systems, sequential consistency is comparable to one-copy serializability by viewing each file or directory operation as a transaction. A file operation executes on a single object, the accessed file, but a directory operation may involve more than one object.

In this section, we prove that our replication control protocol guarantees sequential consistency by demonstrating that it obeys the view change properties and rules of El. Abbadi et al. [1], which they show are sufficient conditions for a replicated database to guarantee one-copy serializability.

We introduce essential definitions, properties and rules in Section 5.1. We present the pseudo code for our protocol in Section 5.2 and prove correctness in Section 5.3.

5.1 Background of View Change Protocol

In this subsection, we introduce the definitions and the theorem proposed by El. Abbadi, et al. [1] for later discussion.

Definitions

In a distributed system that consists of a finite set of processors, a processor's *view* is defined as its estimate of the set of processors with which communication is possible.

Let P be the set of processors, p a member of P, and **P** the power set of P. The function *view:* $P \rightarrow \boldsymbol{P}$ gives the view of each processor p in P.

A *virtual partition* is a set of communicating processors that share a common view and a test for membership in the partition. We assume that at any time, a processor is in at most one virtual partition.

Let V denote the set of all possible virtual partitions. The instantaneous assignment of processors to virtual partitions is given by the partial function *vp:* $P \rightarrow V$. vp is undefined for p if p is not assigned to any virtual partition.

The total function *defview:* $P \rightarrow \{true, false\}$ characterizes the domain of vp, i.e., defview(p) is true if p is currently assigned to some virtual partition, and is false otherwise.

A function *join(p, v)* denotes the event where p changes its local state to indicate that it is currently assigned to v. Similarly, function *depart(p, v)* denotes p changing its local state to indicate that it is no longer assigned to v.

The function *members:* $V \rightarrow \boldsymbol{P}$ yields for each virtual partition v the set of processors that were at some point in their past assigned to v.

El. Abbadi et al. decompose a replication management algorithm into two parts: a replication control protocol that translates each logical operation into one or more physical operations, and a concurrency control protocol that synchronizes the execution of physical operations. Based on this decomposition, they present three properties and five rules, and prove that any replication control protocol that satisfies these properties and rules guarantees one-copy serializability when combined with a concurrency control protocol that ensures conflict-preserving serializability. Below we describe them in turn.

Three Properties

(S1) View consistency

If defview(p) and defview(q) and vp(p)=vp(q), then view(p)=view(q).

(S2) Reflexivity

If defview(p), then $p \in$ view(p).

(S3) Serializability of virtual partitions

For any execution E produced by the replicated data management protocol, the set of virtual partition identifiers occurring in E can be totally ordered by a relation << that satisfies the condition

if v << w and $p \in$ (members(v) $\cap$ view(w)), then depart(p, v) happens before join(q, w) for any $q \in$ members(w).

This property states that p's departure from v must be visible to all the members of its new cohort in w.

Five Rules

(R1) Majority rule

A logical object L is accessible from a processor p assigned to a virtual partition only if a majority of copies of L reside on processors in view(p).

(R2) Read rule

Processor p implements the logical read of L by checking if L is accessible from it and, if so, sending a physical read request to any processor $q \in$ (view(p) $\cap$ copy(L)). (If q does not respond, then the physical read can be retried at another processor or the logical read can be aborted.)

(R3) Write rule

Processor p implements the logical write of L by checking if L is accessible from it and, if so, sending physical write requests to all processors $q \in$ (view(p) $\cap$ copies(L)) which are accessible and have copies of L. (If any physical write request can not be honored, the logical write is aborted).

(R4) Execution rule

All physical operations carried out on behalf of a transaction t must be executed by processors of the same virtual partition v. In this case we say that t *executes* in v.

(R5) Partition initialization rule

Let p be a processor that has joined a new virtual partition v, and let L_p be a copy of a logical object L that is accessible in v. The first operation on L_p must be either a write of L_p performed on behalf of a transaction executing in v, or a *recover(L_p)* operation that writes into L_p *the most recent* value of L written by a transaction executing in some virtual partition u such that u << v for any legal creation order <<.

CP-Serializability

Two physical operations *conflict* if they operate on the same physical object and at least one of them is a write.

An execution E of a set of transactions T is *conflict-preserving (CP) serializable* if there exists an equivalent serial execution E_S of T that preserves the order of execution of conflicting operations.

A concurrency control protocol ensures CP-Serializability if it guarantees that the execution of any set of transactions is conflict-preserving.

THEOREM

Let R be a replication control protocol obeying properties Sl–S3 and rules Rl–R5, and let C be a concurrency control protocol that ensures CP-Serializability of physical operations. Any execution of transactions produced by R and C is one-copy serializability. □

5.2 Protocol Pseudo Code

To demonstrate that our replication control protocol satisfies the properties and rules described above, we give an abstract implementation of the protocol below. For simplicity, we assume a reliable underlying transport protocol, such as TCP, among replication servers. [2]

We specify that the view of a replica *p* for an object *L* is undefined if the replication for *L* is disabled on *p*. We assume each replication server has a unique identifier at its creation time, e.g., its DNS name. A replication server denotes its unique identifier as *myid*. When a replication server starts, its view for each object *L* is initialized as the set of all the replicas holding a copy of *L*, denoted as *L.copies*.

The Boolean function *enabled:* $L \rightarrow \{true, false\}$ is true on a replica *p* if the replication of object *L* on *p* is enabled. On a replica *p*, the variable *L.primary* records the primary server *p* admits for *L*, and the variable *L.view* records *p*'s view for *L* when *L* is accessible on *p*. On a replication server, function *disable(L)* disables the replication of *L*, and function *enable(L)* enables the replication of *L*, respectively.

When a server receives a write-open request from a client, it calls the procedure *ObjectDisable* to disable the replication of the file on other replicas. The server becomes the primary after receiving the acknowledgements from a majority of the replication servers. The routine can fail either due to a network partition that separates the server from the majority of the replicas, or because there is a competing server that starts *ObjectDisable* procedure for *L* simultaneously. In the first case, the server simply keeps trying the process and leaves the client to detect the failure with the request timing out. For the second case, we resolve the conflict with the strategy described in Section 3.1 – the server that disables a majority of the replicas continues; the server that disables fewer replicas quits the procedure; when a tie happens, the server with bigger identifier wins.

In the procedure's pseudo code, the parameter δ in line 6 is an upper bound on the message transmission delay between any two replicas. For a subset $A \subseteq P$, the function *count(A)* (line 10) returns the number of the replicas in A. In line 21, the parameter ε is a random value between [0, 2δ]. We have the server wait for the duration of 2δ+ε if it fails to become the primary, allowing the competitor to disable the replication during this period. The purpose of waiting additional ε time is to resolve extraordinary situations. For example, in a conflict that involves more than two competing servers, having these servers restart the replication disabling processes at different times reduces the likelihood that they conflict again.

[2] The assumption can be easily relaxed by including serial numbers in exchanged messages.

```
procedure ObjectDisable(in L: L);
1   var A: set of P; T: Timer; p, q;
2   while [enabled(L)] do {
3       disable(L); L.contender←myid;
4       for each p ∈ L.view−{myid}     do
5           send(p, "disable", L, myid);
6       T.set(δ×2); A←{myid};
7       while [1] do {
8           select from
9               receive("ok", L, q) ⇒
10                  A←A∪{q};
11                  if [count(A)>count(L.copies)/2] or
[count(A)=count(L.copies)/2 and myid>L.contender]
then
12                      L.primary←myid;
13                      return true;
14                  fi;
15              T.timeout ⇒
16                  for each p ∈ A−{myid} do
17                      send (p, "enable", L, myid, ∅);
18                  enable(L); wait(δ×2+ε);
19                  break;
20          endselect;
21      };
22  };
23  return false;
```

The primary server calls the procedure *ObjectWrite* to update the copies of *L* on other replication servers when it receives a write request for *L*. In the pseudo code, the variable *mtime* records the modification time of *L* specified by the primary server. It is sent to other replication servers along with the update data. Each replica stores this time with its physical copy of *L* respectively.

During *ObjectWrite*, the primary server is the only replica whose view is defined. Therefore it can acknowledge a client write request after a majority of the replicas reply, instead of waiting for the acknowledgements from all the replication servers in its view. This is equal to say that after receiving the acknowledgements from a majority of the replicas, the primary server's view becomes the set of all the replied replicas; the view extends when the acknowledgements from more replication servers are received.

```
procedure ObjectWrite(in L: L, value);
1   var A: set of P; p, q: P;
2       mtime: us (time in microsecond);
3   mtime←current-time; A←{myid};
4   for each p ∈ L.view−{myid} do
5       send(p, "update", L, value, mtime, myid);
6   while [count(A)≤count(L.copies)/2] do {
7       select from
8           receive("ok", L, q) ⇒ A←A∪{q};
9       endselect;
10  };
11  update(L, value, mtime);
12  return;
```

When the client closes *L* after modification, the primary server calls the procedure *ObjectEnable* to enable the replication for *L* on other replication servers. According to NFSv4, a client should not have any pending writes before issuing a close request. This implies that at least a majority of the replication servers have the fresh copy of *L* at this point. However, before re-enabling the replication, the primary server should wait for a period of time, allowing slow replication servers to catch up, as well as constructing a complete active view for *L*. For simplicity, we implement this process with the primary server sending an empty update to other replication servers.

The primary server distributes its view to other active replicas when re-enabling replication. Each of these replication servers stores the view with *L* respectively. Any requests from a replica outside the view are not allowed, until that replica obtains the fresh copy of *L* and is re-added into the view.

```
procedure ObjectEnable(in L: L);
1   var A: set of P; T: Timer; p, q: P;
2   for each p ∈ L.view−{myid} do
3       send(p, "update", L, ∅, L.mtime, myid);
4   T.set(δ×4); A←{myid};
5   while [A≠L.view] do {
6       select from
7           receive("ok", L, q) ⇒ A←A∪{q};
8           T.timeout ⇒
9               if count(A)>count(L.copies)/2 then
10                  break;
11              fi;
12      endselect;
13  };
14  L.primary←∅; enable(L); L.view←A;
15  for each p ∈ L.view−{myid} do
16      send(p, "enable", L, myid, L.view);
17  return;
```

Based on the above procedures, we give the abstract implementation for processing logical read, write-open, write and write-close requests below. In the procedure *LogicalWriteOpen* and *LogicalWriteClose*, we use the variable *L.count* to record the number of concurrent writers for *L* on the primary server. The symbols "<" and ">" delimit critical sections that are protected with a mutual exclusion lock. We note that the *wait* function used in all the procedures automatically releases the lock.

```
procedure LogicalRead(in L: L);

1   if enabled(L) or L.primary=myid then
2       serve client request;
3   else
4       redirect client to L.primary;
5   fi;
```

```
procedure LogicalWriteOpen(in L: L);

1   <if ~enabled(L) and L.primary=myid then
2       L.count++;  acknowledge client request;
3   else if ObjectDisable(L)=true then
4           L.count←1;  acknowledge client request;
5       else if  L.primary≠∅ then
6               redirect client to L.primary;
7           else
8               deny client request;
9           fi;
10      fi;
11  fi;>
```

```
procedure LogicalWrite(in L: L);

1   if enabled(L) or L.primary≠myid then
2       deny client request;
3   else
4       ObjectWrite(L);
5       acknowledge client request;
6   fi;
```

```
procedure LogicalWriteClose(in L: L);

1   <if enabled(L) or L.primary≠myid then
2       deny client request;
3   else
4       L.count−−;  acknowledge client request;
5       if L.count=0 then
6           ObjectEnable(L);
7           if L.view≠L.copies then
8               schedule(Probe(L));
9           fi;
10      fi;
11  fi;>
```

Correctness does not require a failed server to rejoin the active views from which it is excluded, but common sense argues that a repaired server should return to service. Because a rejoining server does not know what it has missed during partition, we charge the primary server with the responsibility of probing for the return of any replication servers that are not in the active view. The primary server performs this by scheduling the task *Probe* after re-enabling the replication (line 8 in the procedure *LogicalWriteClose*).

When detecting a reconnection, the task *Probe* calls the procedure *AddMember* to synchronize the returning replica with the up-to-date copy and re-add that replica to the active view. In the procedure, function *sync(l, source, target)* represents the process that synchronizes the copy of *L* on target with the copy of *L* on source.

```
task Probe(in L: L);

1   var R: set of P; T: Timer; r, q: P;

2   R←L.copies−L.view;
3   while[R≠∅] do {
4       for each r ∈ R do
5           send(r, "probe", myid);
6       T.set(δ×2);
7       while [1] do {
8           select from
9               receive("ok", L, q) ⇒
10                  AddMember(L, q);  reset(T);
11              T.timeout ⇒ break;
12          endselect;
13      };
14      R←L.copies−L.view;
15  };
16  exit task;
```

```
procedure AddMember(in L: L, add-id: P)

1   <if ObjectDisable(L)=true then
2       sync(L, myid, add-id);
3       L.view←L.view∪{add-id};
4       ObjectEnable(L);
5   fi;>
```

If the primary server fails, a connected client can detect the failure when its request times out. At this time, the client appeals another replication server to recover the failure, which triggers the procedure *Recover* on the selected replacement. If the server succeeds in collecting the acknowledgements from a majority of the replication servers, it brings all accessible copies up-to-date, forms a new view and distributes it to the acknowledged replicas. After that, it replies to

the client request and starts the task *Probe* to detect the return of the failed replicas.

```
procedure Recover(in L: L, primary: P)
1   var A: set of P; T: Timer; p, q, sync-id: P;
2       newest: μs (time in microsecond);
3   if Alive(primary) then
4       deny client request; return;
5   fi;
6   if primary∉L.view then
7       acknowledge client request; return;
8   fi;
9   <if enabled(L) then
10      disable(L); L.primary←primary;
11  fi;>
12  while [~enabled(L)] do {
13      <if L.primary≠primary then
14          wait(δ×2); continue;
15      else
16          L.primary←myid; L.contender←myid;
17      fi;>
18      for each p ∈ L.view−{myid} do
19          send(p, "recover", L, myid);
20      T.set(δ×2); A←{myid};
21      newest←L.mtime; sync-id←myid;
22      while [1] do {
23          select from
24              receive("ok", L, mtime_q, q) ⇒
25                  A←A∪{q};
26                  if mtime_q>newest then
27                      newest←mtime_q; sync-id←q;
28                  fi;
29              T.timeout ⇒ break;
30          endselect;
31      };
32      if count(A)>count(L.copies)/2 or [count(A)=
count(L.copies)/2 and myid>L.contender] then
33          if sync-id≠myid then
34              sync(L, sync-id, myid);
35          fi;
36          for each p ∈ A−{myid} do
37              sync(L, myid, p);
38          L.view←A; enable(L);
39          for each p ∈ A−{myid} do
40              send (p, "enable", L, myid, L.view);
41          schedule(Probe(L));
42      else
43          for each p ∈ A−{myid} do
44              send (p, "enable", L, myid, ∅);
45          L.primary←primary; wait(δ×2+ε);
46      fi;
47  };
48  acknowledge client request; return;
```

The task *Monitor* runs on every replication server and is responsible for generating responses to the requests from other replicas. In line 29, the replication server calls the procedure *Alive* to check the state of the original primary server upon receiving a recovery request, so that it does not switch primary blindly under unusual network conditions; e.g. network connections may not be transitive.

```
1   task Monitor;
2   var view: set of P; T: Timer; p: P; L: L;
3       mtime: us (time in microsecond);
4   while[1] do {
5       select from
6           receive("disable", L, p) ⇒
7               if p∉L.view then continue; fi;
8               <if enabled(L) then
9                   disable(L); L.primary←p;
10                  send(p, "ok", myid, L);
11              else if p<L.contender then
12                      L.contender←p;
13                  fi;
14              fi;>
15          receive("update", L, val, mtime, p) ⇒
16              if ~enabled(L) and L.primary=p then
17                  update(L, val, mtime);
18                  send(p, "ok", myid, L);
19              fi;
20          receive("enable", L, p, view) ⇒
21              <if ~enabled(L) and L.primary=p then
22                  enable(L); L.primary←∅;
23                  if view≠∅ then L.view←view; fi;
24                  send(p, "ok", myid, L);
25              fi;>
26          receive("probe", p) ⇒ send(p, "ok", myid);
27          receive("recover", L, p) ⇒
28              if p∉L.view then continue; fi;
29              <if ~Alive(L.primary) then
30                  if enabled(L) then disable(L); fi;
31                  L.primary←p;
32                  send(p, "ok", myid, L, L.mtime);
33              else if p<L.contender then
34                      L.contender←p;
35                  fi;
36              fi;>
37          receive("remove", D, L, mtime, p) ⇒
38              <if ~enabled(D) and D.primary=p and
~enabled(L) and L.primary=p then
39                  remove(D, L, mtime);
40                  send(p, "ok", myid, D);
41              fi;>
42      endselect;
43  };
```

```
procedure Alive(check-id: P);
1   var T: Timer;
2   if check-id=myid then return true; fi;
3   if check-id=∅ then return false; fi;
4   send(check-id, "probe", myid);
5   T.set(δ×2);
6   while [1] do {
7       select from
8           receive("ok", check-id) ⇒ return true;
9           T.timeout ⇒ return false;
10      endselect;
11  };
```

Next we give the entry remove procedure as an example for directory modifications. Unlike file writes, a directory modification can involve more than one object. We disable the replications for all the involved objects before performing a directory update. In the procedure, *D* represents the parent directory of *L* that is to be removed. We show the replication disabling for the two objects as separate operations. In the real implementation, these requests are sent in one message.

```
procedure LogicalRemove(in D, L: L);
1   var A: set of P; T: Timer; p, q: P;
2       mtime: us (time in microsecond);
3   while [~ObjectDisable(D) or ~ObjectDisable(L)]
do
4   {};
5   mtime←current-time;
6   for each p ∈ D.view−{myid} do
7       send(p, "remove", D, L, mtime, myid);
8   T.set(δ×2);  A←{myid};
9   while [A≠D.view] do {
10      select from
11          receive("ok", D, q) ⇒
12              A←A∪{q};
13              if count(A)>count(D.copies)/2 then
14                  remove(D, L, mtime);
15                  acknowledge client;
16              fi;
17          T.timeout  ⇒
18              if count(A)>count(D.copies)/2 then
19                  break;
20              fi;
21      endselect;
22  };
23  D.primary←∅;  enable(D);  D.view←A;
24  for each p ∈ D.view−{myid} do
25      send(p, "enable", D, myid, D.view);
26  if D.view≠D.copies then schedule(Probe(D)); fi;
27  return;
```

5.3 Correctness Proof

To prove that our protocol guarantees sequential consistency, we show that it satisfies properties S1–S3, rules R1–R5, and CP-Serializability.

First, the use of a single replica, the primary server, to determine the view of a majority partition ensures S1. Second, the primary server distributes its view only to the replicas included in the active view, so S2 is guaranteed. Third, every replica departs from its old virtual partition (disable replication) before forming a new partition (obtain a new view and enable replication), which ensures S3.

To see that R1 is satisfied, we observe that an object *L* can be accessed on a server *p* only if the replication of *L* is enabled on *p* or if *p* is the primary server for *L*. In both cases, the view of *p* contains a majority of the replication servers.

In our system, a logical read of object *L* is performed either by reading the physical copy at the connected server *p* when *L* is accessible on *p*, or by reading the copy at the primary server if replication is disabled on *p*. Property S2 then gives us rule R2.

Rule R3, the write rule, follows with similar reasoning. *L* is writable at *p* only if *p* holds a copy of *L* and *p* is the primary server for *L*. *p* then sends the update to all the replication servers in view(p).

Rule R4 is automatically satisfied since a file or directory operation is executed atomically in file systems.

Rule 5 says that before copy l_p can be read in a partition *v*, it must contain the most recent value assigned to *L*. The rule is satisfied in our protocol by requiring each replica to synchronize with the up-to-date copy before being added to the active view (procedure *AddMember*) or forming a new view (procedure *Recover*).

Finally, in our protocol, CP-Serializability is ensured by requiring all access requests to be served from the primary server when concurrent writes occur. □

6. Discussion

As described previously, each replication server stores an active view for every replicated object, which incurs substantial storage overhead. Given our assumption about the relative frequency of failure, it is more efficient to store the complement of an active view on each replication server. In our implementation, we record the complement of an active view in NFSv4 extended attributes, if it is nonempty.

So far, the presented algorithm is sufficient to guarantee sequential consistency. However, when partition occurs, a read operation in a minority partition may

return stale data. Such a situation does not compromise the system state, but it can lead to a loss of external consistency [9], i.e. the ordering of operations inside the system does not agree with the order that an application expects. If sequential consistency is too weak a guarantee for an application, strict consistency that guarantees both one-copy serializability and external consistency is required. Our protocol supports this requirement by disabling data updates everywhere when a failure occurs. We are aware that other methods exist to support strict consistency. For example, replication servers can exchange periodic heartbeat messages to detect partitions and bound the staleness of data, or a reader can check a specified number of replication servers to ensure that it accesses the fresh copy of data. Compared with these methods, our strategy allows less availability for write operations in the case of failure. However, it has two dominant advantages: first, it does not affect read performance, which we believe is critical for a file system to be really useful; second, it adds no overhead or network traffic to normal operations.

7. Related Work

A lot of research has been undertaken on replication control in distributed systems. Our replica control protocol shares many features with the previous work in this area. Specially, our protocol can be regarded as an extension of the primary copy method [6] and a special implementation of the view change protocol in distributed file systems.

Echo [10] and Harp [5] are file systems that use the primary copy scheme to support mutable replication. In these systems, replication is used only to increase data availability; potential performance benefits from replication are not targeted. Both of these systems use pre-determined primary server for a collection of disks, a potential bottleneck if those disks contain hot spots or if the primary server is located remotely from clients. In our system, we avoid this problem by allowing dynamic determination of a primary server, chosen at the granularity of a single file or directory. We use replication to improve performance, as well as availability. A client can choose a nearby or lightly loaded replication server to access data, and switch to a working replication server if the originally selected server fails.

El-Abbadi et al. first proposed a view change protocol in the context of transactional replication systems [1]. Our failure detection and recovery scheme can be regarded as a special implementation of a view change protocol in distributed file systems, with two novelties. First, in our protocol, failure detection and recovery are driven by client accesses. This eliminates the need for periodic heartbeat messages or special group communication services. Second, by taking advantage of the features provided by our primary-copy scheme, when the system is free of failure, our view change protocol is totally embedded into the concurrency control messages (replication enabling messages and replication disabling messages). This helps to reduce the network traffic in normal operations.

Recent years have seen a lot of work in peer-to-peer (P2P) file systems, including OceanStore [15], Ivy [16], Pangaea [19] and Farsite [20]. These systems address the design of systems in untrusted, highly dynamic environments. Consequently, reliability and continuous data availability are usually critical goals in these systems, but performance or data consistency are often sacrificed. Compared to these systems, our system addresses data replication among file system servers, which are more reliable but have more stringent requirements on average I/O performance. This leads to different design strategies in our approach.

8. Conclusion

This paper presents a replication control protocol for distributed file systems that supports strict consistency or sequential consistency, even in partition failures. In the protocol, failure detection and recovery are driven by client accesses. No heartbeat messages or expensive group communication services are required. The protocol imposes a small performance penalty on writes, and no overhead on reads. It is well suited for enterprise computing environments in which reads outnumber writes and failures are rare.

9. References

[1] A. E1 Abbadi, D. Skeen, and F. Cristian, "An Efficient Fault-tolerant Protocol for Replicated Data Management", *Proc. Of 5th ACM SIGACTSIGMOD*, pp. 215-229, (1985).

[2] P. Bernstein and N. Goodman, "The failure and recovery problem for replicated distributed databases", *ACM TODS*, (Dec. 1984).

[3] F. Cristian, H. Aghali, R. Strong and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement", *Proc. Of 15th FTCS*, pp.200-206, (June 1985).

[4] S.B. Davidson, H. GarciaMolina and D. Skeen, "Consistency in Partitioned Networks", *ACM Computing Surveys* **17**(31) (1985).

[5] B. Likov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the Harp File System", *Proc. of 13th SOSP*, Pacific Grove, (Oct. 1991).

[6] P. Alsberg and J. Day, "A Principle for Resilient Sharing of Distributed Resources", *Proc. of 2nd International Conference on Software Engineering*, pp. 627-644, (Oct. 1976).

[7] J. Zhang and P. Honeyman, "Naming, Migration, and Replication in NFSv4", Tech. Report, CITI, University of Michigan, (2003).

[8] F. Cristian and S. Mishra, "Automatic service availability management in asynchronous distributed systems", *Proc. of 2nd International Workshop on Configurable Distributed Systems*, Pittsburgh, PA, (Mar 1994).

[9] Sun Microsystems, Inc., "NFS Version 4 Protocol", RFC 3010 (Dec. 2000).

[10] A. Hisgen, A. Birrel, T. Mann, M. Schroeder, and G. Swart, "Granularity and Semantic Level of Replication in the Echo Distributed File System", *Proc. Of Workshop on Management of Replicated Data*, Houston (Nov. 1990).

[11] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults", *Journal of ACM*, **27** (April 1980).

[12] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM Trans. on Prog. Lang. and Systems* **4**(3) (July 1982).

[13] P.A. Bernstein and N. Goodman, "Concurrency control in distributed database systems", *ACM Computing Surveys.* **13**(2). (1981).

[14] J.N. Gray, "The Transaction Concept: Virtues and Limitations", *Proc. Of 7th VLDB*, pp. 144-154, (1981).

[15] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: the OceanStore Prototype", *Proc. Of 2nd USENIX FAST.* (Mar. 2003).

[16] A. Muthitacharoen, R. Morris, T.M. Gil, and B. Chen, "Ivy: A Read/Write Peer-to-peer File System", *Proc. Of 5th OSDI,* Boston (Dec. 2002).

[17] L. Lamport, "How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", *IEEE Trans. on Computers*, **C-28**(9):690-691, (Sep. 1979).

[18] D.K. Gifford, "Information Storage in a Decentralized Computer System", Tech Report CSL-81-8, Xerox Corporation, (Mar. 1983).

[19] Y. Saito, C. Karamonolis, M. Karlsson, and M. Mahalingam, "Taming aggressive replication in the Pangaea wide-area file system", *Proc .of 5th OSDI,* (Dec. 2002).

[20] A. Adya, W.J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, R.P. Wattenhofer, "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment", *Proc. Of 5th OSDI*, (Dec. 2002).