Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

Brian M. Oki
Barbara H. Liskov

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA 02139

Abstract

One of the potential benefits of distributed systems is their use in providing highly-available services that are likely to be usable when needed. Availability is achieved through replication. By having more than one copy of information, a service continues to be usable even when some copies are inaccessible, for example, because of a crash of the computer where a copy was stored. This paper presents a new replication algorithm that has desirable performance properties. Our approach is based on the primary copy technique. Computations run at a primary, which notifies its backups of what it has done. If the primary crashes, the backups are reorganized, and one of the backups becomes the new primary. Our method works in a general network with both node crashes and partitions. Replication causes little delay in user computations and little information is lost in a reorganization; we use a special kind of timestamp called a viewstamp to detect lost information.

1 Introduction

One of the potential benefits of distributed systems is their use in providing highly-available services, that is, services that are likely to be up and accessible when needed. Availability is essential to many computer-based services; for example, in airline reservation systems the failure of a single computer can prevent ticket sales for a considerable time, causing a loss of revenue and passenger goodwill.

Availability is achieved through replication. By having more than one copy of important information, the service continues to be usable even when some copies are inaccessible, for example, because of a crash of the computer where a copy was stored. Various replication algorithms have been proposed to achieve availability [2, 4, 9, 11, 12, 16, 21, 35]. This paper presents a new replication algorithm that has desirable performance properties.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, and in part by the National Science Foundation under grant DCR-8503662.

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

© 1988 ACM 0-89791-277-2/88/0007/0008 $1.50

Our algorithm runs on a system consisting of nodes connected by a communication network. Nodes are independent computers that communicate with each other only by sending messages over the network. Although both nodes and the network may fail, we assume these failures are not byzantine [24]. Nodes can crash, but we assume they are failstop processors [34]. The network may lose, delay, and duplicate messages, or deliver messages out of order. Link failures may cause the network to partition into subnetworks that are unable to communicate with each other. We assume that nodes eventually recover from crashes and partitions are eventually repaired.

Our replication method assumes a model of computation in which a distributed program consists of *modules*, each of which resides at a single node of the network. Each module contains within it both data objects and code that manipulates the objects; modules can recover from crashes with some of their state intact. No other module can access the data objects of another module directly. Instead, each module provides procedures that can be used to access its objects; modules communicate by means of *remote procedure calls*. Modules that make calls are called *clients*; the called module is a *server*.

Modules are the unit of replication in our method. Ideally, programmers would write programs without concern for availability in some (distributed) programming language that supports our model of computation. The language implementation then uses our technique to replicate individual modules automatically; the resulting programs are highly available.

We assume that computations run as atomic transactions [14]. Our method guarantees the *one-copy serializability* correctness criterion [3, 33]: the concurrent execution of transactions on replicated data is equivalent to a serial execution on non-replicated data.

Our approach is based on the primary copy technique [1, 36], which works roughly as follows. One replica is designated the primary; the others are backups. The primary is responsible for the processing of transactions that use its objects; it notifies the backups of what it has done. When a replica crashes or is separated from the others by a partition, or when a replica recovers from a crash or a partition is repaired, the replicas are reorganized and a new primary is selected if necessary. We refer to this reorganization as a view change [13]. Once the view change is complete, the (new) primary can continue with transaction processing.

The primary copy technique as originally proposed worked only if node failures were distinguishable from network failures; in general such a distinction cannot be made and our method does not require it. In addition, our method exhibits useful performance properties. Transactions encounter little delay in interacting with the replicas, yet little information is lost in a view change. Remote procedure calls to

access or modify objects are executed entirely at the primary, which notifies the backups in background mode. If a view change happens, the effects of a call may or may not survive into the new view. If they do survive, the transaction can commit; otherwise, it must abort. We use a special kind of timestamp called a *viewstamp* to distinguish the two situations.

We begin in the next section with an overview of our method. Sections 3 and 4 describe the two parts of the method, transaction processing and view changes. Section 5 discusses how our method compares with other replication techniques. We conclude in Section 6 with a summary of what we have accomplished.

2 Overview of our Method

The method replicates individual modules to obtain *module groups*. A module group consists of several copies of the module, called *cohorts*, which behave as a single, logical entity; the program can indicate the number of cohorts when the group is created. The set of cohorts is the group's *configuration*. Each cohort has a unique name called an *mid*; the group as a whole bears a unique *groupid*. We expect a small number of cohorts per group, on the order of three or five.

One cohort is designated as the *primary*; it executes procedure calls, and participates in two-phase commit.

The remaining cohorts are *backups*, which are essentially passive and merely receive state information from the primary.

Failures and recoveries are masked when they are noticed. Over time the communication capability within a group may change as cohorts or communication links fail and recover. To reflect this changing situation, each cohort runs in a *view*. A view is a set of cohorts that are (or were) capable of communicating with each other, together with an indication of which cohort is the primary; it is a subset of the configuration and must contain a majority of group members.

A group switches to a new view by executing a *view change protocol*; our protocol is a simplification and modification of the virtual partitions protocol [12]. Each view is identified by a unique *viewid*; we guarantee that viewids are totally ordered. The view change protocol generates a new view and viewid. If a majority of cohorts accept the new view, cohorts switch to the new, *active* view; otherwise, they remain in their old views, but the views become *inactive*. Transactions are processed only in active views.

Views and viewids reflect the current communication patterns, but not the information about committed and active transactions that have run at the group. This additional information is obtained by using *timestamps*. Timestamps are unique within a view and form a total order; they are generated by the primary and are easy to produce, for example, by incrementing a local counter. The primary generates a new timestamp each time it needs to communicate information to its backups; we refer to each such occurrence as an *event*. Examples of events are the completion of processing of a remote call or of a prepare or commit message. Each event is assigned a unique timestamp, and later events receive later timestamps. Instead of checkpointing events directly to the backups, the primary maintains a communication *buffer* (similar to a fifo queue) to which it writes *event records*. An event record identifies the type of the event, and contains other relevant information about the event. Information in the buffer is sent to the backups in timestamp order. The buffer implementation provides reliable delivery of event records to all backups in the primary's view; if it fails to deliver a message, then a crash or communication failure has occurred that will cause a view change.

We use timestamps as an inexpensive way of determining what information survives a view change. Because a timestamp is meaningful only within a view, we introduce *viewstamps*. A viewstamp is simply a timestamp concatenated with the viewid of the view in which the timestamp was generated; we refer to the parts of viewstamp *v* as *v.id* and *v.ts*. Each cohort maintains a *history*, consisting of a sequence of viewstamps, each with a different viewid. We guarantee that for each viewstamp *v* in its history, the cohort's state reflects event *e* from view *v.id* iff *e*'s timestamp is less than or equal to *v.ts*.

The correctness of our algorithm depends on the interaction of transaction processing and the view change algorithm. Transaction processing guarantees that transactions are serialized properly. In addition, it guarantees that a transaction can commit only if all its events are known to at least a majority of cohorts. The view change algorithm guarantees that events known to a majority of cohorts survive into subsequent views. Thus, events of committed transactions will survive view changes. Not all events survive view changes, however; for example, the processing of a particular remote call may be lost. We use the *history* plus some information that arrives in the prepare message to ensure that the transaction will be forced to abort in such a case. On the other hand, if the *history* and the information in the prepare message indicate that all the events associated with the transaction survived the view change, the transaction can commit.

In the next two sections we describe our technique. First, we describe transaction processing and then the view change algorithm.

3 Running Transactions

Our system runs transactions in a manner similar to a system without replication. There are two main differences: we use viewstamps to determine whether a transaction can commit, and instead of writing information to stable storage [25] during two-phase commit, the primary sends it to the backups using the communication buffer discussed above.

The part of a cohort's state that affects transaction processing is summarized in Figure 1. Each cohort has a *status*; if it is "active," it can participate in transaction processing, and otherwise it is involved in a view change. We say that a cohort is *active* if its status is "active"; otherwise it is *inactive*. The *gstate* consists of all objects that constitute the group state. Each *object* has a unique name (relative to the group) and a current value, and also whatever information is needed to implement synchronization and recovery. In

```
status: status              % cohort is active or doing a view change
gstate: {object}            % objects in the group state
mygroupid: int              % the name of the module group
cur_viewid: viewid          % the current viewid
cur_view: view              % the current view
history: [viewstamp]        % indicates events known to cohort
timestamp: int              % the timestamp generator
buffer: [event-record]      % the communication buffer
```

where

```
status = oneof[active, view_manager, underling: null]
object = <uid: int, base: T, lockers: {lock-info}>
lock-info = <locker: aid, info: oneof[read: null, write: T]>
viewid = <cnt: int, mid: int>
view = <primary: int, backups: {int}>
viewstamp = <id: viewid, ts: int>
```

Figure 1: Partial State of a Cohort; {} denotes a set, [ ] denotes a sequence, oneof means a tagged union with component tags and types as indicated, and <> denotes a record, with component names and types as indicated.

the remainder of this paper, we assume that transactions synchronized by means of strict 2-phase locking [18] with read and write locks. Each object has a *base* version of some type T; different objects can be of different types, but we ignore these differences in the paper. A transaction modifies a *tentative* version, which is discarded if the transaction aborts and becomes the base version if it commits. Thus, in addition to its name and base version, an object contains a set of lockers that identifies transactions holding locks on the objects, the kinds of locks held, and any tentative versions created for them.

The primary uses the buffer to communicate information about events to the backups; the implementation of the buffer guarantees reliable delivery of event records to all backups in timestamp order. We distinguish between writing and forcing information to the buffer; a similar distinction is made in transaction systems that use stable storage. Writing means that the information will be delivered to the backups at a convenient time; this is accomplished by calling the *add* operation on the buffer. *Add* takes an event record as an argument. It atomically assigns the event a timestamp (advancing the *timestamp* and updating the *history* in the process) and adds the event record to the buffer; it returns the event's viewstamp. There may be concurrent execution within a module, so the implementation of *add* must serialize the use of the buffer and ensure that event records are recorded in the buffer in timestamp order.

The *force-to* operation is used to force the buffer. Since sometimes it is not necessary to force the entire buffer, the operation takes a viewstamp *v* as an argument. If the viewstamp is not for the current view it returns immediately; otherwise it waits until a *sub-majority* of backups know about all events in the current view with timestamps less than or equal to *v.ts*.[1] A sub-majority is one less than a majority of the configuration; if a sub-majority of backups knows about an event, then a majority of the cohorts in the configuration knows about that event. As mentioned earlier, if a majority of cohorts knows some information, the view change algorithm guarantees that the information will be known in all subsequent views.

Running transactions requires the collaboration of both *clients* and *servers*. Clients create transactions, make any remote calls they contain, and act as coordinators of two-phase commit. Servers process remote calls and participate in two-phase commit; in processing a call, a server may make further calls.

We assume the system provides a highly-available location server that maps groupids to configurations; various implementations are discussed in [15, 20, 22, 31].[2] To find a server it has not used before, a cohort fetches the configuration from the location server and communicates with members of the configuration to determine the current primary and viewid. It stores this information in a local cache.

Below we discuss the work done by active primaries of clients and servers, other processing at cohorts, and processing of query messages. We assume that both clients and servers are replicated; we discuss an alternative to replicating clients in Section 3.5. Our discussion assumes that transactions are one-level; we discuss nested transactions in Section 3.6.

---

[1] *Force-to* delays its caller, but other work, including adding and forcing the buffer, can still go on at the cohort in other processes. If communication with some backups is impossible, the call of *force_to* will be abandoned, and the cohort will switch to running the view change algorithm.

[2] Note that the location server defines the limits of availability; no module group can be more available than it is.

3.1 Active Primaries of Clients

Recall that we intend to use viewstamps to determine whether a transaction can commit. Each time a server finishes processing a remote call on behalf of a transaction, it assigns the call a viewstamp. Information about these viewstamps is collected as the transaction runs in a data structure called the *pset*, which is a set of

<groupid: int, vs: viewstamp>

pairs. The *pset* contains an entry for every call made by the transaction; a pair *<g, v>* indicates that group *g* ran a call for the transaction and assigned it viewstamp *v*.

The processing at the client's primary is summarized in Figure 2. When a transaction is created, it receives a unique transaction identifier *aid* and an empty *pset*. (We make the aid unique across view changes by including *mygroupid* and *cur_viewid* in it.) To make a remote call, the system looks up the primary and viewid for the group in its cache, initializing the cache if necessary, and then sends the call message to the primary. The message contains the viewid from the cache, a unique call id (to prevent duplicate processing of a single call), and information about the call itself (the procedure name and the arguments).

There are three possible results of such a message. The first, and most likely, is a reply message for the call. The reply message contains a *pset* that records *<groupid, viewstamp>* pairs for this call

Starting a transaction:

Create the transaction aid and an empty pset.

Making a remote call:

1. Look up the server in the cache, updating the cache if necessary. Send the call message to the primary; the message contains the unique call id and also the viewid obtained from the cache.
2. If a reply message arrives, add the elements of the pset in the reply message to the transaction's pset. User code at the client can now continue running.
3. If there is no reply, abort the transaction: send abort messages to the participants (determined from the pset), and add an <"aborted", aid> record to the buffer.
4. If the reply indicates that the view has changed, update the cache, if possible, and go to step 1. If a more recent view cannot be discovered, abort the transaction as described above.

Coordinator for two-phase commit:

1. Send *prepare* messages containing the aid and pset to the participants, which can be determined from the pset.
2. If all participants agree to commit, release any local locks held by the transaction and install its tentative versions, add a <"committing", plist, aid> record to the buffer, where the plist is a list of non-read-only participants, and then do a *force-to(new_vs)*, where new_vs is the viewstamp returned by the call on the *add* operation. Send *commit* messages to the participants; when all of them acknowledge the commit, add a <"done", aid> record to the buffer.
3. If there is no answer after repeated tries, update the cache, if possible, and retry the prepare. If a more recent view cannot be discovered, or if any participant refuses to prepare, discard any local locks and versions held by the transaction and send *abort* messages to the participants. Add an <"aborted", aid> record to the buffer.

Figure 2: Processing at the Active Primary of a Client.

and any further remote calls made in processing it. The pairs in the reply's *pset* are added to the transaction's *pset*.

The second possibility is no reply at all (after a sufficient number of probes). In this case, we abort the transaction; we also attempt to update the cache, so that the next use of the server will not cause an abort. The transaction must abort because we cannot know whether the call message would be a duplicate if we sent it to a new primary. The message might be a new one, or it might be a duplicate for a call that ran before the view change or was running when the view change happened. In the first case, we need to do the call; in the second case, we must not redo it. To resolve this uncertainty, we abort the transaction.

The third possibility is a reply indicating that the view has changed. In this case, we update the cache and retry the call. We assume the message delivery system maintains some connection information that enables it to not deliver duplicate messages even in the case when the module crashes and recovers between deliveries. If duplicate messages are possible, we must abort the transaction in this case too.

When the transaction commits, the client's primary acts as the coordinator of the two-phase commit protocol [19]. It determines who the participants are from the *pset*. It sends the *pset* in the prepare messages to allow each participant to determine whether it knows all events of the preparing transaction.

If all participants agree to prepare, the coordinator adds a "committing" record to its buffer and forces the entire buffer to the backups. This ensures that the commit will be known across a view change of the coordinator. The "committing" record lists only the participants where the transaction holds write locks, since only these must take part in phase two; the reply from a participant indicates whether or not it is read-only. Then the coordinator sends commit messages, and, when all are acknowledged, adds a "done" record to the buffer. Note that user code can continue running as soon as the "committing" record has been forced to the backups.

If the transaction aborts, or if any participant refuses the prepare, the coordinator sends abort messages to the participants and adds an "aborted" record to the buffer. This record is not really needed because a view change at the coordinator that leads to a new primary will cause any of the group's transactions to abort automatically. (To avoid such aborts would require some kind of checkpoint mechanism [17].) However, the record is useful for query processing as discussed in Section 3.4.

3.2 Active Primaries of Servers

Servers process remote calls and act as participants in two-phase commit. Each time a call completes, the primary assigns it a viewstamp, and returns this information in the reply message. The primary can agree to prepare only if it knows about all remote calls its group has done on behalf of the preparing transaction. It uses its *history* and the *pset* in the prepare message to determine this.

Processing at the primary of the server is summarized in Figure 3. When the primary receives a call message, it rejects the call if the call's viewid is not equal to *cur_viewid*. Otherwise, it creates an empty *pset* and runs the call, possibly making further nested calls as described above. When the call completes, it adds a "completed-call" record to the buffer; this record identifies each atomic object that was read or written in processing the call, together with the type of lock obtained and the tentative version if any. Then it adds a pair for this call to the call's *pset* and returns the *pset* in the reply message.

When the primary receives a prepare message, it checks whether it knows about all calls made by the transaction to its group by calling *compatible*(*pset, mygroupid, history*):

compatible(ps, g, vh) ≡
  ∀ p ∈ ps (p.groupid = g ⇒
    ∀ v ∈ vh (p.vs.id = v.id ⇒ p.vs.ts ≤ v.ts))

If the *pset* is not compatible with the *history*, it refuses the prepare. Otherwise, it computes the viewstamp of the most recent "completed-call" event by calling *vs_max*(*pset, mygroupid*):

vs_max(ps, g) = p.vs s.t.
  p ∈ ps & p.groupid = g & ∀ p' ∈ ps (p'.groupid = g
    ⇒ p'.vs.id < p.vs.id ∨ (p'.vs.id = p.vs.id & p'.vs.ts ≤ p.vs.ts))

It uses this viewstamp to force the buffer to ensure that all "completed-call" events are known to at least a sub-majority of backups and then sends an acceptance to the coordinator.

When it receives a commit message, the primary forces a "committed" record to the buffer and then sends an acknowledgment to the coordinator. If it receives an abort message, it adds an "aborted" record to the buffer.

3.3 Other Processing at Cohorts

Cohorts that are not active primaries reject messages sent to them by other module groups, except for queries as discussed in the next section. The response to the rejected message contains information about the current viewid and primary if the cohort knows them (for example, if it is a backup in an active view).

Processing a call:

1. If the viewid in the call message is not equal to the primary's cur_viewid, send back a rejection message containing the new viewid and view.
2. Create an empty pset. Then run the call. If it makes any nested calls, process them as described in Figure 2.
3. When the call finishes, add a <"completed-call", object-list, aid> record to the buffer; the object-list lists all objects used by the remote call, together with the type of lock acquired and the tentative version if any. Add a <mygroupid, new_vs> pair to the pset, where new_vs is the viewstamp returned by the call on the *add* operation of the buffer, and send back a reply message containing the pset.

Processing a Prepare Message:

1. If *compatible*(*pset, history, mygroupid*), perform a *force_to*(*vs_max*(*pset, mygroupid*)), release read locks held by the transaction, and then reply *prepared*. In the reply message, indicate whether the transaction held only read locks at this participant. If the transaction is read-only, add a <"committed", aid> record to the buffer.
2. Otherwise, send a message to the coordinator refusing the prepare and abort the transaction: discard its locks and versions and add an <"abort", aid> event record to the buffer.

Processing a Commit Message:

1. Release locks and install versions held by the transaction. Add a <"committed", aid> record to the buffer, do a *force_to*(*new_vs*), where new_vs is the viewstamp return by *add*, and send a *done* message to the coordinator.

Processing an Abort Message:

1. Discard locks and versions held by the aborted transaction and add an <"aborted", aid> record to the buffer.

Figure 3: Processing at the Active Primary of a Server.

Active backups receive messages containing information from the communication buffer. They process event records in timestamp order, updating the state accordingly. The backup can simply store the records, or it can perform them, for example, by setting locks and creating versions for a "completed-call" record. There is a tradeoff here between the amount of processing at the backups, and how much work is needed during a view change before a backup can become a primary. Perhaps a good compromise is to store "completed-call" records (as part of the *gstate*) until the "committed" or "aborted" record for the call's transaction is received; at this point records for a committed transaction would be processed, while those for an aborted transaction would be discarded.

3.4 Queries

Our implementation does not guarantee that all messages about transaction events arrive where they might be needed. For example, if the transaction aborts, we send abort messages to the participants, but do not guarantee they will arrive. Instead, a cohort that needs to know whether an abort occurred sends a query to another cohort that might know. For example, the primary of the participant can send a query to the primary of the coordinator.

To speed up the processing of queries, we allow any cohort to respond to a query whenever it knows the answer. For example, a cohort that is not a primary may know about the abort of a transaction because it received the "aborted" event record from the primary.

3.5 Replicated Clients

The algorithms above assumed that both the client and the server are replicated. It is good to replicate servers, since they do work on behalf of many clients. Replicating a client that is not a server, however, may not be worthwhile.

If the client is not replicated, it is still desirable for the coordinator to be highly available, since this can reduce the "window of vulnerability" [30] in two-phase commit. This can be accomplished by providing a replicated "coordinator-server." The client communicates with such a server when it starts a transaction, and when it commits or aborts the transaction. The coordinator-server carries out two-phase commit as described above on the client's behalf. It also responds to queries about the outcome of the transaction; its groupid is part of the transaction's aid, so that participants know who it is. In answering a query about a transaction that appears to still be active, it would check with the client, but if no reply is forthcoming, it can abort the transaction unilaterally.

3.6 Nested Transactions

The protocol discussed above is quite permissive about when a transaction can prepare, but much less permissive when a client sends a message to a cohort that does not respond. A lack of response causes the entire transaction to abort. Such an abort can cause lots of work to be lost.

Obviously, there are ways to reduce the number of situations in which the abort happens. For example, we could force a special "start call" record to the backups before making a nested remote call. It would be safe to run the call at the new primary if there were no such record, since even if the call ran before the view change, its effects were local to this group and therefore have been undone by the view change. Alternatively, the client could do a probe before making the call to determine the current primary. However, neither of these techniques is satisfactory, since they delay normal processing.

A better approach is to use nested transactions [10, 28, 30]. Nested transactions have two desirable properties. First, they allow concurrency within a transaction in a way that allows the concurrent activities to be serialized. Second, they provide a checkpointing mechanism; if some part of a transaction cannot complete, we can avoid aborting the entire transaction by running that part as a subaction.

Checkpointing is what allows us to minimize the effects of view changes. If the call is made as a subaction, we need not abort the entire transaction if there is no reply. Instead, we can abort just the subaction, and then do the call again as a new subaction. An algorithm for our method in a system with nested transactions is described in [32]; it is based on the implementation of nested transactions in Argus [26, 28].

Subactions are an economical way to cope with view changes. They are not expensive to implement [27]; they are much cheaper than either of the alternatives for avoiding aborts sketched above. Furthermore, we need to abort and redo a call subaction only when the view changes; thus we do extra work only when the problem arises.

3.7 Discussion

There is a one-to-one correspondence between event records and information written to stable storage by a conventional transaction system and therefore our system works because a conventional one does. The "completed-call" records are equivalent to the data records that must be forced to stable storage before preparing, and the "commit" and "abort" records are the same as their stable storage counterparts. The only difference is our treatment of prepares, since we have no prepare record. In a conventional system, the prepare record tells the participant after a crash whether a transaction that ran there before a crash is able to commit. We do not need the prepare record because we use the primary's *history* and the *pset* in the prepare message to determine what to do.

Even when a transaction only has read locks, we must force the "completed-call" records to the backups when preparing to ensure that read locks are held across a view change. A view change may have happened without this primary being aware of it, and there may be a new primary already processing user requests in the other view. Furthermore, the preparing transaction's read-locks may not be known in the new view, so the new primary may allow other transactions to obtain conflicting locks. Forcing the buffer guarantees that the prepare can succeed only if the transaction's locks survived the view change. Without the force, the prepare could succeed at the old primary even though the locks did not survive. In essence, not doing the force is equivalent to not sending the prepare message to a read-only participant; such prepare messages are needed to prevent violations of two-phase locking.

We believe that our method will perform better than a non-replicated system. Remote calls in our system run only at the primary and need not involve the backups and therefore their performance is the same as in a non-replicated system. We expect that *prepare* messages are usually processed entirely at the primary because the needed "completed-call" event records for remote calls of the preparing transaction will already be stored at a sub-majority of cohorts; otherwise, the primary must wait while the relevant part of the buffer is forced to the backups. Careful engineering is needed here to provide both speedy delivery and small numbers of messages. Committing a transaction requires forcing the "committed" record to the coordinator's backups; the remainder of the protocol can run in background. For both preparing and committing, our method will be faster than using non-replicated clients and servers if communication is faster than writing to stable storage, which is often the case provided that the number of backups is small.

4 Changing Views

Transaction processing depends upon forcing information to backups so that a majority of cohorts know about particular events. The job of the view change algorithm is to ensure that events known to a majority of cohorts survive into subsequent views. It does this by ensuring that every view contains at least a majority of cohorts and by starting up the new view in the latest possible state.

If every view has at least a majority of cohorts, then it contains at least one cohort that knows about any event that was forced to a majority of cohorts. Thus we need only make sure that the state of the new view includes what that cohort knows. This is done using viewstamps: the state of the cohort with the highest viewstamp for the previous view is used to initialize the state in the new view. This scheme works because event records are sent to the backups in timestamp order, and therefore a cohort with a later viewstamp for some view knows everything known to a cohort with an earlier viewstamp for that view.

The view change algorithm requires some information to be recorded in the cohort state. This information is summarized in Figure 4, which shows the complete cohort state. Most of this state is volatile and will be lost in a crash; the ramifications of such crashes are discussed in Section 4.2. The exceptions are *mymid*, *configuration*, and *mygroupid*, which are stored on stable storage when the cohort is first created, and *cur_viewid*, which is stored at the end of a view change. When a cohort recovers from a crash, it initializes *up_to_date* to be false, indicating that its *gstate* is not up to date, and initializes *max_viewid* to *cur_viewid*. Then it initializes *status* to be "view_manager"; this causes it to start a view change as discussed below.

Cohorts send periodic "I'm Alive" messages to other cohorts in the configuration. If a cohort notices that it is not communicating with some other cohort in its view, or if it notices that it is communicating with a cohort that it could not communicate with previously, or if it has just recovered from a crash, it initiates a view change. It is the *manager* of this protocol; the other cohorts are the *underlings*.

An overview of the algorithm run by a cohort is shown in Figure 5. The figure shows what the cohort does in each of its three states, "active," "view_manager," and "underling." In the "active" state, the cohort waits for messages to arrive; the **receive** statement selects an arbitrary waiting message for delivery to the program, and dispatches to the arm that matches the name of that message. If the cohort receives a "change" message, this means that the exchange of "I'm alive" messages indicates the need for a view change; it becomes the view manager by changing its status to "view_manager." If it receives an invitation to join a view, and if the new view's viewid is greater than any it has seen so far, it accepts the view and becomes an underling by changing its status to "underling." The procedure *do_accept* records the new viewid in *max_viewid* and sends an acceptance message. There are two kinds of acceptance messages, "normal" ones, and "crashed" ones. If the cohort is up to date (i.e., *up_to_date* = *true*), it sends an acceptance containing its current viewstamp and an indication of whether it is the primary in the current view. Otherwise, it sends a "crash-accept" response; this response contains only its viewid, and means that it has forgotten its gstate.

If it is a view manager, the cohort sends invitations to join the new view to all other cohorts, and waits for responses. The procedure *make_invitations* creates a new viewid by pairing *mymid* with a number greater than *max_viewid.cnt* and stores it in *max_viewid*. Notice that the new viewid will be different from any produced by another cohort. Then it sends invitations containing *max_viewid* to the other cohorts, records its own response ("crashed" or "normal"), and collects the other responses. If an invitation with a higher viewid arrives, it signals *invited*, returning the new viewid and the mid of the inviter. In this case, the view manager accepts the invitation and

```
status: status              % cohort is active or doing a view change
gstate: {object}            % objects in the group's state
up_to_date: bool            % true if gstate is meaningful
configuration: {int}        % modules in the configuration
mymid: int                  % name of this module
mygroupid: int              % name of the group
cur_viewid: viewid          % current viewid
cur_view: view              % current view
history: [viewstamp]        % indicates events known to cohort
max_viewid: viewid          % highest viewid seen so far
timestamp: int              % the timestamp generator
buffer: [event_record]      % the communication buffer

where

view = status = oneof[active, view_manager, underling: null]
object = <uid: int, base: T, lockers: {lock-info}>
lock-info = <locker: aid, info: oneof[read: null, write: T]>
viewid = <cnt: int, mid: int>
view = <primary: int, backups: {int}>
viewstamp = <id: viewid, ts: int>
```

Figure 4: State of a Cohort.

```
while true do
 tagcase status

  tag active:
   receive    % accept a message
    when change: status := view_manager
    when invite (vid: viewid, m: mid):
     if vid ≤ max_viewid then continue end  % ignore the msg
     do_accept(vid, m)
     status := underling
    others: % transaction messages handled here
    end  % receive

  tag view_manager:
   responses := make_invitations( )
    except when invited (vid: viewid, m: mid):
     do_accept(vid, m)
     status := underling
     continue  % continue at next iteration
     end except
   v: view := form_view(responses)
    except when cannot: continue end  % wait and then try again
   if v.primary = mymid
    then start_view(v)
     status := active
    else send init_view(max_viewid, v) to v.primary
     status := underling
    end  % if

  tag underling:
   await_view( )
    except
     when timeout:
      status := view_manager
      continue
     when invited (vid: viewid, m: mid):
      do_accept(vid, m)
      continue
     when become_primary(v: view): start_view(v)
     end  % except
   status := active

 end  % tagcase
end  % while
```

Figure 5: The View Change Algorithm.

becomes an underling. Otherwise, when all cohorts accept the invitation or a timeout expires, *make_invitations* returns the responses. In this case, the view manager attempts to form a new view (the details are discussed below). If the attempt fails, (*form_view* signals *cannot*), the cohort attempts another view formation later. If the attempt succeeds, and if the view manager is not the new primary, it sends an "init_view" message to the new primary, and becomes an underling. Otherwise it starts the new view: it updates *cur_view* and *cur_viewid*, stores zero in *timestamp* and appends *<cur_viewid, 0>* to the *history*, and writes *cur_viewid* to stable storage. Then it initializes the buffer to contain a single "newview" event record; this record contains *cur_view*, *history*, and *gstate*. Finally, it becomes active.

View formation can succeed only if two conditions are satisfied: at least a majority of cohorts must have accepted the invitation, and at least one of them must know all forced information from previous views. The latter condition may not be true if some acceptances are of the "crashed" variety. For example, suppose there are three cohorts, *A*, *B* and *C*, and that view *v1 = <primary: A, backups: {B, C}>*. Suppose that *A* committed a transaction, forcing its event records to *B* but not *C*, then *A* crashed and recovered, and then a partition occurred that separated *B* from *A* and *C*. In this case we cannot form a new view until the partition is repaired because *A* has lost information and there are forced events that *C* does not know.

The correct rule for view formation is: a majority of cohorts have accepted and

1. a majority of cohorts accepted normally, or
2. crash_viewid < normal_viewid, or
3. crash_viewid = normal_viewid and the primary of view normal_viewid has done a normal acceptance of the invitation.

Here *crash_viewid* is the largest viewid returned in a "crashed" acceptance, and *normal_viewid* is the largest viewstamp returned in a "normal" acceptance. Condition (1) says we can ignore crashed acceptances if we have enough normal ones; condition (2) says we can ignore crashed acceptances if they are from old views; and condition (3) says we can ignore a crashed acceptance if we have information from the primary of its view, because the primary always knows at least as much as any backup.

If the view can be formed, the cohort returning the largest viewstamp (in a "normal" acceptance) is selected as the new primary; the old primary of that view is selected if possible, since this causes minimal disruption in the system.

A cohort in the underling state calls *await_view* to wait to find out what happened to the new view. If no message arrives within some interval, *await_view* signals *timeout* and the cohort becomes the view manager and attempts to form another view. If an invitation for a higher viewid arrives, *await_view* signals *invited*, and the cohort accepts the invitation. If an "init_view" message containing a viewid equal to *max_viewid* arrives, *await_view* signals *become_primary*; the cohort initializes itself to be a primary as discussed above, and becomes active. If a "newview" record for a view with viewid equal to *max_viewid* arrives from the buffer, *await_view* initializes the cohort state before returning: it initializes *cur_view*, *cur_viewid*, *history* and *gstate* from the information in the message, writes *cur_viewid* to stable storage, sets *up_to_date* to true (to indicate that it now has information in *gstate*), and returns normally. Then the cohort becomes active.

4.1 Discussion

When failures or recoveries are detected by the system, the view change protocol runs in each affected module group. The protocol requires relatively little message-passing in the simple case of no additional failures and no concurrent view managers. One round of messages is all that is needed when the manager is also the primary in the last active view; otherwise, one round plus one message is needed.

The system performs correctly even if there are several active primaries. This situation could arise when there is a partition and the old primary is slow to notice the need for a view change and continues to respond to client requests even after the new view is formed. The old primary will not be able to prepare and commit user transactions, however, since it cannot force their effects to the backups.

If the same cohort is the primary both before and after the view change, then no user work is lost in the change. Otherwise, we guarantee the following: Transactions that prepared in the old view will be able to commit, and those that committed will still be committed. Transactions that had not yet prepared before the change may be able to prepare afterwards, depending on whether the completion events of the remote calls are known in the new view. Aborts of transactions may have been forgotten, but delivery of abort messages is not guaranteed in any case; recovery from lost messages is done by using queries (see Section 3.4). To minimize disruption while a view change is happening, or when there is no active view, queries can be answered by any cohort that knows the answer.

The algorithm is tolerant to several cohorts simultaneously acting as managers; the one that chooses the higher viewid will ultimately succeed. Having several managers will slow things down, since there will be more message traffic, but the slow down will be slight. Furthermore, we can avoid concurrent managers to some extent by various policies. For example, the cohorts could be ordered, and a cohort would become a manager only if all higher-priority cohorts appear to be inaccessible.

However, the algorithm is not tolerant of lost messages and slow responses. For example, suppose a manager waits only until it hears from a sub-majority even though there are other cohorts that could respond. This would result in those other cohorts being excluded from the new view, which in turn will mean another round of view changing will occur shortly. If that next view change also excludes some potential members, that will lead to another view change, and so on.

To avoid such a situation, a manager should use a fairly long timeout while it waits to hear from all cohorts that the "I'm alive" messages indicate should reply. Similarly, an underling should use a fairly long timeout before it becomes a manager. In addition, it is worthwhile to mask lost messages by sending duplicates, so that a lost message won't trigger another view change.

A final point is that not all view changes described above really need to be done. One special case is when an active primary notices that it cannot communicate with a backup, but it still has a sub-majority of other backups. In this case, the primary can unilaterally exclude the inaccessible backup from the view. Similarly, an active primary can unilaterally add a backup to its view. View changes are really needed only when the primary is lost, or when a current active view loses enough members that it is no longer a majority. In the latter case, we need not do a view change either; we make the primary inactive since this stops it from working on transactions when it will not be able to commit them.

4.2 Stable Storage

In our algorithm we assumed that most of a cohort's state was volatile. Such an assumption means that if a majority of cohorts are crashed "simultaneously," we may lose information about the module group's state. Here we view a cohort as crashed if either it is really crashed, or if it has recovered from a crash, but its *up_to_date*

variable is false. Note that a catastrophe does not cause a group to enter a new view missing some needed information. Rather, it causes the algorithm to never again form a new view.

Whether it is worthwhile to worry about catastrophes depends on the likelihood of occurrence and the importance of the information in the group state. The considerations here are similar to decisions about when it is necessary to store information in stable storage in a nonreplicated system, except that replication makes the probability of catastrophe smaller to begin with.

If protection against catastrophes is desired, there are various techniques that could be tried. For example, we might use stable storage only at the primary or we might supply each cohort with a universal power supply and have them write information to nonvolatile storage in the background.

5 Related Work

In this section we discuss the relationship of our approach to other work on replication and view changes.

The best known replication technique is voting [16, 21]. With voting, write operations are usually performed at all cohorts, and reads are performed at only one cohort, but in general writes can be performed at a majority of cohorts and reads at enough cohorts that each read will intersect each write at at least one cohort. The write all/read one choice is preferred when reads are much more common than writes.

Our method is faster than voting for write operations since we require fewer messages. Also, we avoid the deadlocks that can arise if messages for concurrent updates arrive at the cohorts in different orders. Our method will also be faster for read operations if these take place at several cohorts. If reads take place at just one cohort, voting may outperform our method because reading can occur at any cohort, while reading in our scheme must happen at the primary, which could become a performance bottleneck. On the other hand, the real source of a bottleneck is a node, not a cohort, and we can organize our system so that primaries of different groups usually run on different nodes. Furthermore, the system can be configured to place primaries at more powerful nodes most of the time. This organization could lead to better performance than voting.

Voting allows operations to continue running as long as the needed number of cohorts are up and accessible. However, when writes must happen at all cohorts, the lost of a single cohort can cause writes to become unavailable. The *virtual partitions protocol* [12, 13] was invented to solve this problem. Our view change protocol is a simplification and modification of this protocol and has better performance. The virtual partitions protocol requires three phases. The first round establishes the new view, the second informs the cohorts of the new view, and in the third, the cohorts all communicate with one another to find out the current state. We avoid extra work by using viewstamps in phase 1 (the first round) to determine what each cohort knows. Our technique can be used in conjunction with voting when writes are done at all members of a view. Just as we use viewstamps, in such a system timestamps assigned when transactions commit could be used to determine which replica has the most information about transaction commits (the timestamps would not contain information about the state of active transactions). Systems in which writes only go to a majority are more difficult to optimize in this way since there is usually no cohort whose state contains at least as much information as the state of any other cohort.

Virtual partitions force transactions that were active across a view change to abort. For example, a transaction that did a remote call in the old view will not be able to prepare in the new view. We use viewstamps to avoid the abort and we rely on the fact that knowledge of later events implies knowledge of earlier ones. A total order on viewstamps would be costly to implement with voting, since there is no single place (like our primary) to generate the viewstamp. It might be possible to use multipart viewstamps [23, 29], however. This is a matter for future research.

A different approach to replication is taken in Isis [4, 5]. Isis works only in a local area network because its view change protocol does not tolerate partitions. In Isis, calls are sent to a single cohort. If the called procedure is a read, the cohort acquires a read lock locally and performs the operation locally. If the procedure is a write, the cohort acquires write locks at all cohorts before doing the call. (Write locks are acquired using a two-phase algorithm that prevents deadlocks in the case of concurrent writes.) Then the cohort performs the call. In either case, the cohort communicates the effects of reads[3] and writes to other cohorts in background mode, and piggybacks them on reply messages. This piggybacked information accompanies all future client messages, including calls to other servers as well as prepare and commit messages. This means, for example, that if the prepare message is sent to a different cohort from the one that performed the call, the information about the effect of the call will be present at the cohort doing the prepare, so there will be no need for that cohort to wait for the background message to arrive, and no possibility that it would need to reject the prepare. Unlike our *pset*, however, piggybacked information in Isis cannot be discarded when transactions commit. A disadvantage of Isis is the large amount of extra information flowing on every message, and the difficulty in garbage collecting that information.

Our method avoids these problems at the cost of a possible delay at prepare time (to force the buffer) and of an occasional abort when there is a view change. The viewstamps in our method represent the information flowing in Isis. However, since the viewstamps only indicate that certain events have occurred, but not what these events are, we must sometimes wait for information about its events to arrive in buffer messages. Also, we must sometimes abort a transaction because information about its events is lost in a view change.

In Cooper's replicated remote procedure calls [9], each procedure call is replicated and executed at every cohort of a server. This technique has high overhead during normal system operation: it requires lots of messages, is wasteful of computation, and requires that programs be deterministic. The advantage of the method is that recovery is inexpensive.

Finally, Tandem's NonStop system [2, 7, 8] and the Auragen system [6] are primary copy methods but there is just one backup, so they can survive only a single failure. Furthermore, the primary/backup pair must reside at a single node (containing multiple processors). If these constraints are acceptable, these methods are efficient. Ours is more general.

6 Conclusions

This paper has described a new replication method for providing high availability. The method performs well in the normal case, does view changes efficiently, and loses little information in a view change. We expect the performance of our method to be comparable to that of a system in which modules are not replicated and better than most other replication methods. At present we are implementing our method; we will be able to run experiments about system performance when our implementation is complete.

Our view change algorithm is highly likely not to lose work in a view change. If a transaction's effects are known at the new primary,

[3]The effect of a read is that a read lock has been acquired.

the transaction can commit. Our notion of viewstamps allows us to determine inexpensively how much each cohort knows and whether a transaction can be committed. Our policy of choosing the primary of the last active view to be the new primary whenever possible avoids losing work altogether; even remote calls that were running before the view change can continue to run afterwards. Note that the probability of aborts can be decreased further if desired. There is a tradeoff here between loss of information in view changes and speed of processing calls. For example, if "completed call" records were forced to the backups before the call returned, there would be no aborts due to view changes, but calls would be processed more slowly.

Choosing the primary of the old view to be the new primary minimizes information loss and makes the view change protocol run quickly. On the other hand, we could modify the protocol to always choose a particular cohort to be the primary if possible. Such a policy matches the needs of some applications. The policy would not cause loss of information: if the old primary is a member of the new view, all its events will survive into the new view. However, work in progress at the old primary would be lost in the change (unless some additional mechanism is included); this includes aborting transactions for which the primary is the coordinator. In addition, a few extra messages will sometimes be needed in the view change protocol.

We presented our algorithm in a system with one-level transactions. However, as noted earlier, such a system can lead to aborts in which a substantial amount of work can be lost. The problem arises when a client gets no reply for a remote call; the transaction must be aborted to avoid running a call more than once. Nested transactions prevent the abort of the top level transaction, and, furthermore, do so efficiently.

In defining our algorithm, we chose to avoid the use of stable storage as much as possible because we were interested in understanding the extent to which having several replicas eliminated the need for stable storage. We found that catastrophes (loss of a group's state) that would not happen if events were recorded on stable storage could sometimes occur in our system. The probability of a catastrophe depends on the configuration, e.g., on whether the cohort's nodes are failure independent. The algorithm can be modified in various ways to reduce the probability of catastrophe if it is considered to be too high.

The use of viewstamps is an interesting compromise between loss of work in failures and extra information. Isis represents one extreme here: no work is lost when there is a failure but large amounts of information must flow around the system. Other systems have no information like viewstamps and must abort all transactions affected by a failure.

Viewstamps may also be worthwhile in a nonreplicated system. In such a system, records containing the effects of calls could be written to stable storage in background mode; the records, like event records, would contain viewstamps. When the prepare message arrives, it would only be necessary to force the records; no delay would be encountered if the records had already been written. A crash would not cause active transactions to abort automatically; instead, queries would be sent to coordinators to determine the outcomes. The result would be a system that is more tolerant of crashes (by avoiding aborts) and also faster at prepare time.

Acknowledgments

We are thankful for the helpful comments of readers of earlier drafts of this paper, and especially to Sanjay Ghemawat, Dave Gifford, Bob Gruber, Deborah Hwang, Elliot Kolodner, Gary Leavens, Sharon Perl, Liuba Shrira, and Bill Weihl.

References

1. Alsberg, P. A., and Day, J. D. A Principle for Resilient Sharing of Distributed Resources. Proc. of the 2nd International Conference on Software Engineering, October, 1976, pp. 627-644. Also available in unpublished form as CAC Document number 202 Center for Advanced Computation University of Illinois, Urbana-Champaign, Illinois 61801 by Alsberg, Benford, Day, and Grapa.

2. Bartlett, J. F. A NonStop Kernel. Proc. of the 8th ACM Symposium on Operating System Principles, SIGOPS Operating System Review, 15 5, December, 1981, pp. 22-29.

3. Bernstein, P. A., and Goodman, N. The Failure and Recovery Problem for Replicated Databases. Second ACM Symposium on the Principles of Distributed Computing, August, 1983, pp. 114-122.

4. Birman, K. P., Joseph, T. A., Rauchle, T., and El Abbadi, A. "Implementing Fault-tolerant Distributed Objects". *IEEE Trans. on Software Engineering 11*, 6 (June 1985), 502-508.

5. Birman, K. P. and Joseph, T. A. "Reliable Communication in the Presence of Failures". *ACM Trans. on Computer Systems 5*, 1 (February 1987), 47-76.

6. Borg, A., Baumbach, J., and Glazer, S. A Message System Supporting Fault Tolerance. Proc. of the 9th ACM Symposium on Operating System Principles, SIGOPS Operating System Review, 17, 5, October, 1983, pp. 90-99.

7. Borr, A. J. Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing. Proc. of the Seventh International Conference on Very Large Data Bases, September, 1981, pp. 155-165.

8. Borr, A. J. Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach. Proc. of the Tenth International Conference on Very Large Data Bases, August, 1984, pp. 445-453.

9. Cooper, E. C. Replicated Distributed Programs. UCB/CSD 85/231, University of California, Berkeley, CA, May, 1985.

10. Davies, C. T. "Data Processing Spheres of Control". *IBM Systems Journal 17*, 2 (February 78), 179-198.

11. Eager, D. L., and Sevcik, K. C. "Achieving Robustness in Distributed Database Systems". *ACM Trans. on Database Systems 8*, 3 (September 1983), 354-381.

12. El Abbadi, A., Skeen, D., and Cristian, F. An Efficient, Fault-Tolerant Protocol for Replicated Data Management. Proc. of the 4th ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems, 1985.

13. El Abbadi, A., and Toueg, S. Maintaining Availability in Partitioned Replicated Databases. Proc. of the 5th ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems, 1986.

14. Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. "The Notions of Consistency and Predicate Locks in a Database System". *Comm. of the ACM 19*, 11 (November 1976), 624-633.

15. Fowler, R. J. Decentralized Object Finding Using Forwarding Addresses. 85-12-1, University of Washington, Dept. of Computer Science, Seattle, WA, December, 1985.

16. Gifford, D. K. Weighted Voting for Replicated Data. Proc. of the 7th ACM Symposium on Operating Systems Principles, SIGOPS Operating Systems Review, 13, 5, December, 1979, pp. 150-162.

17. Gifford, D. K.; and Donahue, J. E. Coordinating Independent Atomic Actions. Proc. of IEEE CompCon85, February, 1985, pp. 92-95.

18. Gray, J. N., Lorie, R. A. Putzolu, G. F., and Traiger, I. L. Granularity of locks and degrees of consistency in a shared data base. In *Modeling in Data Base Management Systems*, G. M. Nijssen, Eds., Elsevier North-Holland, New York, 1976, pp. 365-394.

19. Gray, J. N. Notes on Database Operating Systems. In *Lecture Notes in Computer Science 60*, Goos and Hartmanis, Eds., Springer-Verlag Berlin, 1978, pp. 393-481.

20. Henderson, C. Locating Migratory Objects in an Internet. M.I.T. Laboratory for Computer Science, Cambridge, MA, 1983.

21. Herlihy, M. P. "A Quorum-Consensus Replication Method for Abstract Data Types". *ACM Trans. on Computer Systems 4*, 1 (February 1986), 32-53.

22. Hwang, D. J. Constructing a Highly-Available Location Service for a Distributed Environment. Technical Report MIT/LCS/TR-410, M.I.T. Laboratory for Computer Science, Cambridge, MA, January, 1988.

23. Ladin, R., Liskov, B., and Shrira, L. A Technique for Constructing Highly-Available Services. M.I.T. Laboratory for Computer Science, Cambridge, MA, January, 1988. To be published in *Algorithmica*.

24. Lamport, L., Shostak, R., and Pease, M. "The Byzantine Generals Problem". *ACM Trans. on Programming Languages and Systems 4*, 3 (July 1982), 382-401.

25. Lampson, B. W., and Sturgis, H. E. Crash Recovery in a Distributed Data Storage System. Xerox Research Center, Palo Alto, Ca., 1979.

26. Liskov, B., and Scheifler, R. "Guardians and Actions: Linguistic Support for Robust Distributed Programs". *ACM Trans. on Programming Languages and Systems 5*, 3 (July 1983), 381-404.

27. Liskov, B., Curtis, D., Johnson, P., and Scheifler, R. Implementation of Argus. Proc. of the Eleventh ACM Symposium on Operating Systems Principles, SIGOPS Operating Systems Review, 21, 5, November, 1987, pp. 111-122.

28. Liskov, B. "Distributed Programming in Argus". *Comm. of the ACM 31*, 3 (March 1988), 300-312.

29. Liskov, B., and Ladin, R. Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Colletion. Proc. of the Fifth ACM Symposium on the Principles of Distributed Computing, August, 1986.

30. Moss, J. E. B. Nested Transactions: An Approach to Reliable Distributed Computing. Technical Report MIT/LCS/TR-260, M.I.T. Laboratory for Computer Science, June, 1981.

31. Mullender, S., and Vitanyi, P. Distributed Match-Making for Processes in Computer Networks---Preliminary Version. Proc. of the Fourth Symposium on the Principles of Distributed Computing, ACM, August, 1985.

32. Oki, B. M. *Viewstamped Replication for Highly-Available Distributed Systems.* Ph.D. Th., Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, May 1988. Forthcoming.

33. Papadimitriou, C. H. "Serializability of Concurrent Database Updates". *J. of the ACM 24*, 4 (October 1979), 631-653.

34. Schneider, F. B. Fail-Stop Processors. Digest of Papers from Spring CompCon '83 26th IEEE Computer Society International Conference, March, 1983, pp. 66-70.

35. Skeen, D., and Wright, D. D. Increasing Availability in Partitioned Database Systems. TR 83-581, Dept. of Computer Science, Cornell University, March, 1984.

36. Stonebraker, M. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES". *IEEE Trans. on Software Engineering 5*, 3 (May 1979), 188-194.