Providing High Availability Using Lazy Replication

[1]

Rivka Ladin
Digital Equipment Corp.
One Kendall Square
Cambridge, MA 02139

Barbara Liskov
Liuba Shrira
Sanjay Ghemawat
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

To provide high availability for services such as mail or bulletin boards, data must be replicated. One way to guarantee consistency of replicated data is to force service operations to occur in the same order at all sites, but this approach is expensive. For some applications a weaker causal operation order can preserve consistency while providing better performance. This paper describes a new way of implementing causal operations. Our technique also supports two other kinds of operations: operations that are totally ordered with respect to one another, and operations that are totally ordered with respect to all other operations. The method performs well in terms of response time, operation processing capacity, amount of stored state, and number and size of messages; it does better than replication methods based on reliable multicast techniques.

This research was supported in part by the National Science Foundation under Grant CCR-8822158 and in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under Contract N00014-89-J-1988.

---

[1]A preliminary version of this paper appeared in the Proc. of the Ninth ACM Symposium on Principles of Distributed Computing, August 1990.

1. Introduction

Many computer-based services must be *highly available*: they should be accessible with high probability despite site crashes and network failures. To achieve availability, the server's state must be *replicated*. Consistency of replicated state can be guaranteed by forcing service operations to occur in the same order at all sites. However, some applications can preserve consistency with a weaker, causal ordering [19], leading to better performance. This paper describes a new technique that supports causal order. An operation call is executed at just one replica; updating of other replicas happens by *lazy* exchange of "gossip" messages — hence the name "lazy replication". The replicated service continues to provide service in spite of node failures and network partitions.

We have applied the method to a number of applications, including distributed garbage collection [18], deadlock detection [8], and orphan detection [23], locating movable objects in a distributed system [15], and deletion of unused versions in a hybrid concurrency control scheme [35]. Another system that can benefit from causally-ordered operations is the familiar electronic mail system. Normally, the delivery order of mail messages sent by different clients to different recipients, or even to the same recipient, is unimportant, as is the delivery order of messages sent by a single client to different recipients. However suppose client *c1* sends a message to client *c2* and then a later message to *c3* that refers to information in the earlier message. If, as a result of reading *c1*'s message, *c3* sends an inquiry message to *c2*, *c3* would expect its message to be delivered to *c2* after *c1*'s message. Therefore, read and send mail operations need to be causally ordered.

Applications that use causally-ordered operations may occasionally require a stronger ordering. Our method allows this. Each operation has an ordering type; in addition to the causal operations, there are forced and immediate operations. *Forced operations* are performed in the same order (relative to one another) at all replicas. Their ordering relative to causal operations may differ at different replicas but is consistent with the causal order at all replicas. They would be useful in the mail system to guarantee that if two clients are attempting to add the same user-name on behalf of two different users simultaneously, only one would succeed. *Immediate operations* are performed at all replicas in the same order relative to all other operations. They have the effect of being performed immediately when the operation returns, and are ordered consistently with external events [11]. They would be useful to remove an individual from a classified mailing-list "at once", so that no messages addressed to that list would be delivered to that user after the remove operation returns.

It is easy to construct and use highly-available applications with our method. The user of a replicated application just invokes operations and can ignore replication and distribution, and also the ordering types of the operations. The application programmer supplies a non-replicated implementation, ignoring complications due to distribution and replication, and defines the category of each operation (e.g., for a mail service the designer would indicate that send_mail and read_mail are causal, add_user and delete_user are forced, and delete_at_once is immediate). To determine the operation categories, the programmer can use techniques developed for determining permissible concurrency [14, 31, 32, 34].

Our method does not delay update operations (such as send_mail), and typically provides the response to a query (such as read_mail) in one message round trip. It will perform well in terms of response time, amount of stored state, number of messages, and availability in the presence of node and communication failures provided most

update operations are causal. Forced operations require more messages than causal operations; immediate operations require even more messages and can temporarily slow responses to queries. However, these operations will have little impact on overall system performance provided they are used infrequently. This is the case in the mail system where sending and reading mail is much more frequent than adding or removing a user.

Our system can be instantiated with only forced operations. In this case it provides the same order for all updates at all replicas, and will perform similarly to other replication techniques that guarantee this property (e.g., voting [10] or primary copy [28]). Our method generalizes these other techniques because it allows queries to be performed on stale data while ensuring that the information observed respects causality.

The idea of allowing an application to make use of operations with differing ordering requirements appears in the work on ISIS [3] and Psync [27], and in fact we support the same three orders as ISIS. These systems provide a reliable multicast mechanism that allows processes within a process group consisting of both clients and servers to communicate; a possible application of the mechanism is a replicated service. Our technique is a replication method. It is less general than a reliable multicast mechanism, but is better suited to providing replicated services that are distinct from clients and as a result provides better performance: it requires many fewer messages than the process-group approach, the messages are smaller, and it can tolerate network partitions.

Our technique is based on the gossip approach first introduced by Fischer and Michael [9], and later enhanced by Wuu and Bernstein [36] and our own earlier work [16]. We have extended the earlier work in two important ways: by supporting causal ordering for updates as well as queries and by incorporating forced and immediate operations to make a more generally applicable method. The implementation of causal operations is novel and so is the method for combining the three types of operations.

The rest of the paper is organized as follows. Section 2 describes the implementation technique. The following sections discuss system performance, scalability, and related work. We conclude with a discussion of what we have accomplished.

2. The Replicated Service

Lazy replication is intended for an environment in which individual computers, or *nodes*, are connected by a communication network. Both the nodes and the network may fail, but we assume the failures are not Byzantine. The nodes are fail-stop processors. The network can partition, and messages can be lost, delayed, duplicated, and delivered out of order. The configuration of the system can change; nodes can leave and join the network at any time. We assume nodes have loosely synchronized clocks. There are practical protocols, such as NTP [26] that with low cost synchronize clocks in geographically distributed networks.

A replicated application is implemented by *service* consisting of replicas running at different nodes in a network. To hide replication from clients, the system also provides *front end* code that runs at client nodes. To call an operation, a client makes a local call to the front end, which sends a call message to one of the replicas. The replica executes the requested operation and sends a reply message back to the front end. Replicas communicate new information (e.g., about updates) among themselves by lazily exchanging *gossip* messages.

The are two kinds of operations: *update* operations modify but do not observe the application state, while *query* operations observe the state but do not modify it. (Operations that both update and observe the state can be treated as an update followed by a query.) When requested to perform an update, the front end returns to the client immediately and communicates with the service in the background. To perform a query, the front end waits for the response from a replica, and then returns the result to the client.

We assume there is a fixed number of replicas residing at fixed locations and that front ends and replicas know how to find replicas; a technique for reconfiguring services (i.e., adding or removing replicas) is described in [15]. We also assume that replicas eventually recover from crashes; Section 3.2 discusses how this happens.

In this section we describe how the front end and service replicas together implement the three types of operations. Section 2.1 describes the implementation of causal operations; Section 2.2 extends the implementation to support the other two types of operations.

2.1. Causal Operations

In our method, the front end informs the replicas about the causal ordering of operations. Every reply message for an update operation contains a unique identifier, *uid*, that names that invocation. In addition, every (query and update) operation *o* takes a set of uids as an argument; such a set is called a *label*.[2] The label identifies the updates whose execution must precede the execution of *o*. In the mail service, for example, the front end can indicate that one send_mail must precede another by including the uid of the first send_mail in the label passed to the second. Finally, a query operation returns a value and also a label that identifies the updates reflected in the value; this label is a superset of the argument label, ensuring that every update that must have preceded the query is reflected in the result. Service replicas thus perform the following operations:

update (prev: label, op: op) **returns** (uid: uid)
query (prev: label, op: op) **returns** (newl: label, value: value)

where *op* describes that actual operation to be performed (i.e., gives its name and arguments).

A specification of the service for causal operations is given in Figure 2-1. We view an execution of a service as a sequence of events, one event for each update and query operation performed by a front end on behalf of a client. At some point between when an operation is called and when it returns, an event for it is appended to the sequence. An event records the arguments and results of its operation. In a query event *q*, *q.prev* is the input label, *q.op* defines the query operation, *q.value* is the result value, and *q.newl* is the result label; for update *u*, *u.prev* is the input label, *u.op* defines the update operation, and *u.uid* is the uid assigned by the service. If *e* is an event in execution sequence E, P(e) denotes the set of events preceding *e* in E. Also, for a set S of events, S.label denotes the set of uids of update events in S.

The specification describes the behavior of queries, since this all the clients (via the front ends) can observe. The first clause states that all updates identified by a query result label correspond to calls made by front ends. The second clause states that the result label identifies all required updates plus possibly some additional ones. The third

[2] A similar concept occurs in [3] but no practical way to implement it was described.

Let q be a query. Then

1. q.newl $\subseteq$ P(q).label.
2. q.prev $\subseteq$ q.newl.
3. u.uid $\in$ q.newl $\Rightarrow$ for all updates v s.t. dep(u, v), v.uid $\in$ q.newl.
4. q.value = q.op (Val (q.newl)).

Figure 2-1: Specification of the Causal Operations Service.

clause states that the returned label is *dependency complete*: if some update operation *u* is identified by the label, then so is every update that *u* depends on. An update *u depends on* an update *v* if it is constrained to be after *v*:

dep(u, v) $\equiv$ (v.uid $\in$ u.prev)

The dependency relation *dep* is acyclic because front ends do not create uids.

The fourth clause defines the relationship between the value and the label returned by a query: the result returned must be computed by applying the query q.op to a state Val arrived at by starting with the initial state and performing the updates identified by the label in an order consistent with the dependency relation. If dep(u, v), v.op is performed before u.op; operations not constrained by dep are performed in arbitrary order. Note that this clause guarantees that if the returned label is used later as input to another query, the result of that query will reflect the effects of all updates observed in the earlier query.

The front end maintains a label for the service. It guarantees causality as follows:

1. Client calls to the service. The front end sends its label in every call message and merges the uid or label in each reply message with its label by performing a union of the two sets.
2. Client-to-client communication. The front end intercepts all messages exchanged between its client and other clients. It adds its label to each message its client sends and merges the label in each message received by its client with its label.

The resulting order may be stronger than needed. For example, if client C1 communicates with C2 without exposing any information about its earlier calls on service operations, it is not necessary to order C2's later calls after C1's earlier ones. Our method allows a sophisticated client to use uids and labels directly to implement the causality that really occurs; we assume in the rest of the paper, however, that all client calls go through the front end.

During normal operation a front end will always contact the same "preferred" replica. However, if the response is slow, it might send the request to a different replica or send it to several replicas in parallel. In addition, messages may be duplicated by the network. In spite of these duplicates, update operations must be performed at most once at each replica. To allow the service to recognize multiple requests for the same update, the front end associates a unique call identifier, or *cid*, with each update. That cid is included in every message sent by the front end on behalf of that update.

2.1.1. Implementation Overview

For the method to be efficient, we need a compact representation for labels and a fast way to determine when an operation is ready to be executed. In addition, replicas must be able to generate uids independently. All these properties are provided by a single mechanism, the *multipart timestamp*. A multipart timestamp *t* is a vector

$$t = < t_1, \ldots , t_n >$$

where *n* is the number of replicas in the service. Each part is a nonnegative integer counter, and the initial (zero) timestamp contains zero in each part. Timestamps are partially ordered in the obvious way:

$$t \leq s \equiv (t_1 \leq s_1 \wedge \ldots \wedge t_n \leq s_n)$$

Two timestamps *t* and *s* are *merged* by taking their component-wise maximum. (Multipart timestamps were used in Locus [30] and also in [9, 13, 16, 22, 36].)

Both uids and labels are represented by multipart timestamps. Every update operation is assigned a unique multipart timestamp as its uid. A label is created by merging timestamps; a label timestamp *t* identifies the updates whose timestamps are less than or equal to *t*. The dependency relation is implemented as follows: if an update *v* is identified by *u.prev* then *u* depends on *v*. Furthermore, if *t* and *t'* are two timestamps that represent labels, $t \leq t'$ implies that *t* identifies a subset of the updates identified by *t'*.

A replica receives *call messages* from front ends and also *gossip messages* from other replicas. When a replica receives a call message for an update it hasn't seen before, it processes that update by assigning it a timestamp and appending information about it to its log. The log also contains information about updates that were processed at other replicas and propagated to this replica in gossip messages. A replica maintains a local timestamp, *rep_ts*, that identifies the set of records in the its log and thus expresses the extent of its knowledge about updates. It increments its part of the timestamp each time it processes an update call message; therefore, the value of the replica's part of *rep_ts* corresponds to the number of updates it has processed. It increments other parts of *rep_ts* when it receives gossip from other replicas. The value of any other part *i* of *rep_ts* counts the number of updates processed at replica *i* that have propagated to this replica via gossip.

A replica executes updates in dependency order and maintains its current state in *val*. When an update is executed, its uid timestamp is merged into the timestamp *val_ts*, which is used to determine whether an update or query is ready to execute; an operations *op* is ready if its label $op.prev \leq val_ts$. The replica keeps track of the cids of updates that have been executed in the set *inval*, and uses the information to avoid duplicate executions of updates. Since the same update may be assigned more than one uid timestamp (because the front end sent the update to several replicas), the timestamps of duplicates for an update are merged into *val_ts*. In this way we can honor the dependency relation no matter which of the duplicates a front end knows about (and therefore includes its uid in future labels).

Figure 2-2 summarizes the state at a replica. (In the figure, { } denotes a set, [ ] denotes a sequence, oneof means a tagged union with component tags and types as indicated, and < > denotes a record, with components and types as indicated.) In addition to information about updates, the log also contains information about acks; acks are discussed below.

```
node: int              % replica's id.
log: {log-record}      % replica's log
rep_ts: mpts           % replica's multipart timestamp
val: value             % replica's view of service state
val_ts: mpts           % timestamp associated with val
inval: {cid}           % updates that participated in computing val
ts_table: [mpts]       % ts_table(p)= latest multipart timestamp received from p

  where

log-record = < msg: op-type, rnode: int, ts: mpts >
op-type = oneof [ update: < prev: mpts, op: op, cid: cid, time: time >, ack: < cid: cid, time: time > ]
mpts = [ int ]
```

Figure 2-2: The state of a replica.

The description above ignored two important implementation issues, controlling the size of the log and the size of *inval*. An update record can be discarded from the log as soon as it is known everywhere and has been reflected in *val*. In fact, if an update is known to be known everywhere, it will be ready to be executed and therefore will be reflected in *val*, for the following reason. A replica knows some update record *u* is known everywhere if it has received gossip messages containing *u* from all other replicas. Each gossip message includes enough of the sender's log to guarantee that when the receiver receives record *u* from replica *i*, it has also received (either in this gossip message or an earlier one) all records processed at *i* before *u*. Therefore, if a replica has heard about *u* from all other replicas, it will know about all updates that *u* depends on, since these must have been performed before *u* (because of the constraint on front ends not to create uids). Therefore, *u* will be ready to execute.

The table *ts_table* is used to determine whether a log record is known everywhere. Every gossip message contains the timestamp of its sender; *ts_table(k)* contains the largest timestamp this replica has received from replica *k*. Note that the current timestamp of replica *k* must be at least as large as the one stored for it in *ts_table*. If $ts_table(k)_j = t$ at replica *i*, then replica *i* knows that replica *k* has learned of the first *t* update records created by replica *j*. Every record *r* contains the identity of the replica that created it in field *r.node*. Replica *i* can remove update record *r* from its log when it knows that *r* has been received by every replica, i.e., when

$$\text{isknown}(r) \equiv \forall \text{ replicas } j,\ \text{ts_table}(j)_{r.node} \geq r.ts_{r.node}$$

holds at *i*.

The second implementation issue is controlling the size of *inval*. It is safe to discard a cid *c* from *inval* only if the replica will never attempt to apply *c's* update to *val* in the future. Such a guarantee requires an upper bound on when messages containing information about *c's* update can arrive at the replica. A front end will keep sending messages for an update until it receives a reply. Since reply messages can be lost, replicas have no way of knowing when the front end will stop sending these messages unless it informs them. The front end does this by sending an acknowledgment message *ack* containing the cid of the update to one or more of the replicas. In most applications, separate ack messages will not be needed; instead, acks can be piggybacked on future calls. Acks are added to the log when they arrive at a replica and are propagated to other replicas in gossip the same way updates are propagated.

(The case of volatile clients that crash before their acknowledgements reach the replicas can be handled by adding client crashcounts to update messages and propagating crash information to replicas; a higher crash count would serve as an ack for all updates from that client with lower crash counts.)

Even though a replica has received an ack, it might still receive a message for the ack's update since the network can deliver messages out of order. We deal with late messages by having each ack and update message contain the time at which it was created. Each time the front end sends an update or ack message, it includes in the message the current time of its node's clock; if multiple messages are sent for an update, they will contain different times. The time in an ack must be greater than or equal to the time in any message for the ack's update. An update message *m* is discarded because it is "late" if

m.time + δ < the time of the replica's clock

where δ is greater than the anticipated network delay. Each ack *a* is kept at least until

a.time + δ < the time of the replica's clock

After this time any messages for the ack's update will be discarded because they are late.

A replica can discard a cid *c* from *inval* as soon as an ack for *c's* update is in the log and all records for *c's* update have been discarded from the log. The former condition guarantees a duplicate call message for *c's* update will not be accepted; the latter guarantees a duplicate will not be accepted from gossip (see Section 2.1.4 for a proof). A replica can discard ack *a* from the log once *a* is known everywhere, the cid of *a's* update has been discarded from *inval*, and all call messages for *a's* update are guaranteed to be late at the replica. Note that we rely here on clocks being monotonic. Typically clocks are monotonic both while nodes are up and across crashes. If they need to be adjusted this is done by slowing them down or speeding them up. Also, clocks are usually stable; if not the clock value can be saved to stable storage [21] periodically and recovered after a crash.[3]

For the system to run efficiently, clocks of server and client nodes should be loosely synchronized with a skew bounded by some ε. Synchronized clocks are not needed for correctness, but without them certain suboptimal situations can arise. For example, if a client node's clock is slow, its messages may be discarded even though it just sent them. The delay δ should be chosen to accommodate both the anticipated network delay and the clock skew. The value for this parameter can be as large as needed because the penalty of choosing a large delay only affects how long servers remember acks. Note that we do not require reliable delivery within δ; instead the front ends mask delivery failures by re-sending messages.

It may seem that our reliance on synchronized clocks affects the availability of the system. A problem could arise only if a node's clock fails, the node is unable to carry out the clock synchronization protocol because of communication problems, and yet the node is able to communicate with other nodes in the system. Such a situation is extremely unlikely.

---

[3]Writes to stable storage can be infrequent; after a crash, a node must wait until its clock is later than the time on stable storage + β, where β is a bound on how frequently writes to stable storage happen, before communicating with replicas.

2.1.2. Processing at Each Replica

This section describes the processing of each kind of message. Initially, *rep_ts* and *val_ts* are zero timestamps, *ts_table* contains all zero timestamps, *val* has the initial value, and the log and *inval* are empty. The log and *inval* are implemented as hash tables hashed on the cid.

Processing an update message:

Replica *i* discards an update message *u* from a front end if it is late (i.e., if *u.time* + δ < the time of the replica's clock) or it is a duplicate (i.e., its cid *c* is in *inval* or a record *r* such that *r.cid* = *u.cid* is in the log). If the message is not discarded, the replica performs the following actions:

1. Advances its local timestamp *rep_ts* by incrementing its i^{th} part by one while leaving all the other parts unchanged.
2. Computes the timestamp for the update, *ts*, by replacing the i^{th} part of the input argument *u.prev* with the i^{th} part of *rep_ts*.
3. Constructs the update record *r* associated with this execution of the update,

   r := makeUpdateRecord(*u, i, ts*)

   and adds it to the local log.
4. Executes *u.op* if all the updates that *u* depends on have already been incorporated into *val*. If *u.prev* ≤ *val_ts*, then:

   val := apply(*val, u.op*) % perform the op
   val_ts := merge(*val_ts, r.ts*)
   inval := *inval* ∪ {*r.cid* }
5. Returns the update's timestamp *r.ts* in a reply message.

The *rep_ts* and the timestamp *r.ts* assigned to *u* are not necessarily comparable. For example, *u* may depend on update *u'*, which happened at another replica *j*, and which this replica does not know about yet. In this case $r.ts_j > rep_ts_j$. In addition, this replica may know about some other update *u"* that *u* does not depend on, e.g., *u"* happened at replica *k*, and therefore, $r.ts_k < rep_ts_k$.

Processing a query message:

When replica *i* receives a query message *q*, it compares the query's input label, *q.prev*, with *val_ts*, which identifies all updates reflected in *val*. If *q.prev* ≤ *val_ts*, it applies *q.op* to *val* and returns the result and *val_ts*. Otherwise, it waits since it needs more information. It can either wait for gossip messages from the other replicas or it might send a request to another replica to elicit the information. *Val_ts* and *q.prev* can be compared part by part to determine which replicas have the missing information.

Processing an ack message:

A replica processes an ack as follows:

1. Advances its local timestamp *rep_ts* by incrementing the i^{th} part of the timestamp by one while leaving all the other parts unchanged.
2. Constructs the ack record *r* associated with this execution of the ack:

   r := makeAckRecord(*a, i, rep_ts*)

   and adds it to the local log.
3. Sends a reply message to the front end.

Note that ack records do not enter *inval*.

Processing a gossip message:

A gossip message contains *m.ts*, the sender's timestamp, and *m.new*, the sender's log. The processing of the message consists of three activities: merging the log in the message with the local log, computing the local view of the service state based on the new information, and discarding records from the log and from *inval*.

When replica *i* receives a gossip message *m* from replica *j*, it discards *m* if *m.ts* ≤ *j's* timestamp in *ts_table*. Otherwise, it continues as follows:

1. Adds the new information in the message to the replica's log:

   $log := log \cup \{ r \in m.new \mid \neg(r.ts \leq rep_ts) \}$

2. Merges the replica's timestamp with the timestamp in the message so that *rep_ts* reflects the information known at the replica:

   rep_ts := merge(*rep_ts*, *m.ts*)

3. Inserts all the update records that are ready to be added to the value into the set *comp*:

   $comp := \{ r \in log \mid \text{type}(r) = \text{update} \wedge r.prev \leq rep_ts \}$

4. Computes the new value of the object:

   while *comp* is not empty **do**
   select *r* from *comp* s.t. ∃ no *r'* ∈ *comp* s.t. *r'.ts* ≤ *r.prev*
   $comp := comp - \{ r \}$
   if $r.cid \notin inval$ **then**
   $val := \text{apply}(val, r.op)$
   $inval := inval \cup \{ r.cid \}$
   $val_ts := \text{merge}(val_ts, r.ts)$

5. Updates *ts_table*:

   $ts_table(j) := m.ts$

6. Discards update records from the log if they have been received by all replicas:

   $log := log - \{ r \in log \mid \text{type}(r) = \text{update} \wedge \text{isknown}(r) \}$

7. Discards records from *inval* if an ack for the update is in the log and there is no update record for that update in the log:

   $inval := inval - \{ c \in inval \mid \exists\, a \in log \text{ s.t. type}(a) = \text{ack} \wedge a.cid = c \wedge$
   $\exists \text{ no } r' \in log \text{ s.t. type}(r') = \text{update} \wedge r'.cid = c \}$

8. Discards ack records from the log if they are known everywhere and sufficient time has passed and there is no update for that ack in *inval*:

   $log := log - \{ a \in log \mid \text{type}(a) = \text{ack} \wedge \text{isknown}(a) \wedge a.time + \delta < \text{replica local time} \wedge$
   $\exists \text{ no } c \in inval \text{ s.t. } c = a.cid \}$

The new value can be computed faster by first sorting *comp* such that record *r* is earlier than record *s* if *r.ts* ≤ *s.prev*.

Since the decision to delete records from the log uses information from all other replicas, there may be a problem during a network partition. For example, suppose a partition divided the network into sides A and B and *r* is known at all replicas in A and also at all replicas in B. If no replica in A knows that *r* is known in B, there is nothing we can do. However, as suggested in [36], progress can be made if replicas include their copy of *ts_table* in gossip messages and receivers merge this information with their own *ts_tables*. In this way, each replica would get a more recent view of what other nodes know.

2.1.3. Optimizations

The size of gossip messages can be reduced by not sending records the receiver already knows. Furthermore, it is not necessary to send information that another replica is likely to send. For example a sender might include in gossip only the records it created in response to requests it received from the client and that the recipient doesn't know; a replica could request other records if necessary, e.g., if the originating replica isn't communicating with it right now.

The number of gossip messages can be reduced substantially by arranging the replicas in a communication structure such as a spanning tree. Each replica would send gossip only to its neighbors. If there is a failure (crash or partition), the structure would be reconstituted by carrying out a view change algorithm [6, 7]. This approach causes information to propagate more slowly than having replicas gossip with all other replicas.

Communication between the front end and the replicas can be made more efficient by taking advantage of the fact that a front end typically communicates with the same replica. Communication could be done over a streaming connection such as TCP or Mercury [24]. In this case, the front end need not wait to receive the uid timestamp from an update it requested before sending the next operation (query or update) to the replica. Instead, the replica maintains a copy of the front end's timestamp, into which it merges uids of updates as they complete. Furthermore, the front end's timestamp need be included in a message to the replica only if it increases because the front end received a timestamp in a message from another client. (A similar optimization is used in ISIS [4].) Streaming does not cause responses to queries to be delayed. Client-to-client communication may be delayed, however; the front end cannot send on a message from its client to another client until it knows the timestamp for the most recent update requested by its client.

A further optimization is possible when using streaming: operations from the front end can be batched if they are small. (Mercury streams [24] do this automatically.) A message would be sent from the front end when the client does a query, when the buffer is full, or when its client is sending a message to another client. This technique will cause an additional delay only in the latter case; the front end may need to flush the stream and wait for a reply before sending on the client message. The reply from the replica need contain only one timestamp — the one that would have been included in the reply to the last request in the batch. Batching will be most effective in applications containing a large number of updates relative to queries, or when clients are able to continue doing other work while a query is being processed (so that queries can be batched too).

The saving in timestamps that is possible with streaming can also apply to information in the log and in gossip. For example, a sequence of updates from a single front end can be associated with the timestamp of the first update in the sequence; the receiving replica can compute the timestamps for the other updates by incrementing the sender's part of the timestamp for each of them.

2.1.4. Analysis

In this section we argue informally that the implementation is correct and makes progress, and that entries are removed from the log and *inval* eventually. The discussion considers the protocol described in Section 2.1.2 and ignores the optimizations described in Section 2.1.3.

The specification in Figure 2-1 defines a centralized service in which each update is performed only once and is assigned a single uid. However, the implementation is distributed and a single update may be processed several times at the different replicas and may thus be assigned several different uids. We will show that in spite of the duplicates, the implementation satisfies the specification, i.e., as far as client can tell from the information received from queries, each update is executed only once.

The implementation uses timestamps to represent both uids and labels. As far as uids are concerned, we require only uniqueness, and this is provided by the way the code assigns timestamps to updates. Several timestamps may correspond to the same update; these correspond to duplicate requests that arrived at different replicas and were assigned different timestamps. For labels, timestamps provide a compact way of representing a set of uids: a label timestamp t identifies an update u if there exists a record r for u such that $r.ts \leq t$.

Correctness: The first clause of the specification requires that only updates requested by front ends are executed by the service. It follows from the code of the protocol, which only creates update timestamps in response to update messages from front ends. The second clause requires that the updates identified by the query input label *q.prev* also be identified by the query output label *q.newl*. It follows from the timestamp implementation of labels and from the query code, which returns only when $q.prev \leq val_ts$. The third clause requires that the label *q.newl* be dependency complete. It follows from the timestamp implementation of uids and labels and from the update processing code, which guarantees that if u depends on v then there exists a record r for v such that $r.ts \leq u.prev$ and therefore the set of updates identified by a label timestamp is trivially dependency complete.

The fourth clause requires that *q.value* be the result of applying the query *q.op* to the state derived by evaluating the set of updates identified by *q.newl* in an order consistent with the dependency relation. We begin by establishing several lemmas, each concerning the state variables of a single replica. We assume in the proofs that each operation is performed atomically at a single replica and that gossip is processed in a single atomic step.

Lemma 1: After an ack record a enters the log at a replica, no duplicate of *a's* update will be accepted from the front end or network at that replica.

Proof: By inspection of the code we know that after an ack record a enters a replica's log, the following holds: a is in the log or a left the log at a point when $a.time + \delta <$ the time of the replica's clock. If a message for *a's* update arrives later, it will be discarded by the update processing code if a is in the log, and otherwise it will be rejected because it is late, assuming the front end guarantees that an update message contains an earlier time than any ack message for its update, and the replica's clock is monotonic. □

Lemma 2: After an update record r enters the log at a replica, no duplicate of the update will be

accepted from the front end or network at that replica.

Proof: By inspection of the code we know that after a record *r* enters the log at a replica, the following holds: *r* is in the log, or *r's* cid has entered *inval*, or *r's* cid has left *inval* but at that point an ack for *r* was in the log. The update processing code and Lemma 1 ensure that these conditions are sufficient to eliminate all future duplicates of *r*, whether these duplicates are created by the network or by the front end. □

Lemma 3: Replica *i* has received the first *n* records processed at replica *k* if and only if the k^{th} part of replica *i's* timestamp is greater than or equal to *n*, i.e., $rep_ts_k \geq n$.

Proof: First note that part *i* of replica *i's* timestamp *rep_ts* counts the number of front end update messages processed at *i* that entered *i's* log. A record in *i's* log is transmitted by gossip to other replicas until it is deleted from the log. A record *r* is deleted from the log only when *isknown(r*) holds at *i*, i.e., when *i* knows *r* has reached all other replicas. Therefore, each replica knows a prefix of every other replica's log. Since the gossip timestamp is merged into the timestamp of the receiving replica, it is easy to see that part *j*, $i \neq j$, of replica *i's* timestamp counts the number of records processed at *j* that have been brought by gossip to replica *i*. □

Lemma 4: If *isknown(r)* holds at replica *i*, all duplicate records for *r's* update have arrived at *i*.

Proof: Recall that a replica *i* knows that all replicas have received an update record *r* when it has received a gossip message containing *r* from each replica. But at this point it has also received from each replica *j* all the records processed by *j* before receiving *r*. Therefore, at this point it has received all duplicates of *r* that were processed at other replicas before they received *r*. By Lemma 2, no duplicate will be accepted from the front end or network at a replica after receiving *r*. Therefore, *i* must have received all duplicates of *r* at this point. □

Lemma 5: If *isknown(r)* holds at replica *i*, all duplicate records *d* for *r's* update have $d.ts \leq rep_ts$.

Proof: By Lemma 4 we know that all duplicates of *r's* update *u* have arrived at this replica. Furthermore, records for all updates that *u* depends on are also at this replica because front ends do not manufacture uids: *u.prev* can only contain uids generated by the service, so any update whose timestamp was merged into *u.prev* must have been processed at a replica before that replica knew about update record *r*, and therefore when all replicas have sent gossip containing *r* to replica *i*, they have also sent records for all updates that *u* depends on. Now, let *r'* be either *r*, a duplicate of *r*, or a record for an update that *u* depends on and let *k* be the replica where *r'* was created. By Lemma 3, we know that $rep_ts_k \geq r'.ts_k$. Since this is true for all such *r'*, $rep_ts \geq d.ts$ for all duplicates *d*. □

Lemma 6: When a record *r* for an update *u* is deleted from the log, *u* is reflected in the value.

Proof: When *r* is deleted from the log at replica *i*, *isknown(r*) holds at *i* and therefore by Lemma 5, $r.ts \leq rep_ts$. This implies that $r.prev < rep_ts$. Therefore *r* enters the set *comp* and either *u* is executed, or *r's* cid is in *inval* and therefore *u* was executed earlier. □

Lemma 7: At any replica $val_ts \leq rep_ts$.

Proof: It is easy to see that the claim holds initially. It is preserved by the update processing code

because if an update is executed, only field *i* of *val_ts* changes and $val_ts_i = rep_ts_i$. It is preserved by gossip processing because for each record *r* in *comp*, $r.prev \leq rep_ts$. Since *r.ts* differs from *r.prev* only in field *j*, where *r* was processed at *j*, and since *r* is known locally, $r.ts_j \leq rep_ts_j$ by Lemma 3. □

Lemma 8: For any update *u*, if there exists a record *r* for *u* s.t. $r.ts \leq rep_ts$, *u* is reflected in the value.

Proof: The proof is inductive. The basis step is trivial since there is no record with a zero timestamp. Assume the claim holds up to some step in the computation and consider the next time *rep_ts* is modified. Let *r* be an update record for *u* s.t. $r.ts \leq rep_ts$ after that step. We need to show that *u* is reflected in the value. First, consider a gossip processing step. Since $r.ts \leq rep_ts$, we know $u.prev \leq rep_ts$. If *r* is in the log, it enters *comp* and either *u* is executed now or *u's* cid is in the set *inval* and therefore *u* is already reflected in the value. If *r* is not in the log, then $r.ts \leq rep_ts$ implies (by Lemma 3) that *r* was deleted from the log and by Lemma 6, *u* is already reflected in the value. Therefore we have shown that *u* is reflected in the value after a gossip processing step. Now consider an update processing step. If $r.ts \leq rep_ts$ before this step, the claim holds by the induction assumption. Otherwise, we have $\neg(r.ts \leq rep_ts)$ before this step and $r.ts \leq rep_ts$ after the message was processed. In the processing of an update, replica *i's* timestamp *rep_ts* increases by one in part *i* with the other parts remaining unchanged. Therefore, the record being created in this step must be *r* and furthermore $u.prev \leq rep_ts$ before this step occurred. Therefore, any *v* that *u* depends on has already been reflected in the value by the induction assumption, and $u.prev \leq val_ts$. Therefore *u* is reflected in the value in this step. □

We are now ready to prove the fourth clause of the specification. Recall that a query returns *val* and *val_ts*. Lemmas 7 and 8 guarantee that for any update record *r* such that $r.ts \leq val_ts$, *r's* update is reflected in the value *val*. Therefore, all updates identified by a query output label are reflected in the value. We will now show that the updates are executed only once and in the right order.

To prove that updates are executed only once at any particular replica, we show that after an update *u* is reflected in the value, it will not be executed again. The cid *c* that entered the set *inval* when *u* was executed must be deleted from *inval* before *u* can be executed again. However, when *c* is deleted from *inval* no duplicate record for *u* is present in the log and an ack for *c's* update is present in the log. By Lemma 1, the presence of the ack guarantees that no future duplicate from the front end or the network will reenter the log. Furthermore, when *c* is deleted from *inval*, *isknown(r*) holds for some update *r* for *c*, so by Lemma 4, all duplicates *d* of *r* have arrived at the replica. By Lemma 5, $d.ts \leq rep_ts$ and therefore step 1 of the gossip processing code ensures that any future duplicate *d* arriving in a gossip message will not reenter the replica's log.

We now show that updates are reflected in the value in an order consistent with the dependency relation. Consider an update record *r* such that $r.ts \leq val_ts$ and an update *v* such that *r's* update *u* depends on *v*. We need to show that *v* is reflected in the value before *u*. From the implementation of the dependency relation we know there exists an update record *s* for *v* such that $u.prev \geq s.ts$. Therefore, by Lemmas 7 and 8 both *u* and *v* are reflected in *val*. Consider the first time *u* is reflected in the value. If this happens in the processing of an update message, at that step $u.prev \leq val_ts$; by Lemma 7, $u.prev \leq rep_ts$, and therefore by Lemma 8 *v* has already been reflected in *val*. If this happens while processing a gossip message, a record for *u* has entered the set *comp* and so $u.prev \leq rep_ts$ and therefore $s.ts \leq rep_ts$. By Lemma 3, *s* has entered the log at this replica and will enter *comp* now unless it has

already been deleted. The code guarantees that when both *s* and a record for *u* are in *comp* either *v* is reflected before *u* because *u.prev* ≥ *s.ts*, or *v's* cid is in *inval* and so *v* was reflected earlier. If *s* was deleted from the log earlier, then by Lemma 6 *v* has already been reflected.

Making Progress: To ensure progress, we need to show that updates and queries indeed return. It is easy to see that updates return provided replicas eventually recover from crashes and network partitions are eventually repaired. These assumptions also guarantee that gossip messages will propagate information between replicas. Therefore, from the gossip processing code and Lemma 3, replica and value timestamps will increase and queries will eventually return.

Garbage Collection: Update records will be removed from the log assuming replica crashes and network partitions are repaired eventually. Also, acks will be deleted from the log assuming crashes and partitions are repaired eventually and replica clocks advance, provided cids are deleted from *inval*. To show that cids are deleted from *inval*, we need the following lemma, which guarantees that an ack stays in the log long enough to prevent its cid from reappearing in *inval*:

Lemma 9: *Isknown(a)* ∧ type (*a*) = ack ⇒ *d.ts* ≤ *rep_ts* for all duplicates *d* of *a's* update.
Proof: Similar to Lemma 5. □

Lemmas 8 and 9 and the ack deletion code guarantee that an ack is deleted only after its cid is deleted from *inval*. By Lemma 1, no duplicates of the update message from the front end or network arriving after this point will be accepted assuming the time in the ack is greater than or equal to the time in any message for the update. Furthermore, step 1 of gossip processing ensures that duplicates of the update record arriving in gossip will not enter the replica's log. Therefore cids will be removed from *inval* assuming crashes and partitions are repaired eventually and front ends send acks with big enough times.

Availability: The service uses the query input label to identify the requested updates so it is important that the label identify just the required updates and no others. However, labels in fact do sometimes identify extra updates. For example, consider two independent updates *u* and *v* with *u.prev* = *v.prev* and assume that *v* is processed at replica *i* before *u*. This means that *r.ts* > *s.ts*, where *r* and *s* are the update records created by *i* for *u* and *v*, respectively. When *r.ts* is merged into a label L, L also identifies *v* as a required update. Nevertheless, a replica never needs to delay a query waiting for such an "extra" update to arrive because the gossip propagation scheme ensures that whenever the "required" update record *r* arrives at some replica, all update records with timestamps smaller than *r's* will be there. Note that the timestamp of an "extra" update record is always less than the timestamp of some "required" update record identified by a label.

2.2. Other Operation Types

The section shows how the implementation can be extended to support forced and immediate updates. These two stronger orders are provided only for updates; queries continue to be ordered as specified by their input labels. As mentioned, each update operation has a declared ordering type that is defined when the replicated application is created. Recall that forced updates are performed in the same order (relative to one another) at all replicas; immediate updates are performed at all replicas in the same order relative to all other updates.

Like causal updates, forced updates are ordered via labels and uids but their uids are totally ordered with respect to one another. Labels now identify both the causal and forced updates, and the input label for an update or query identifies the causal and forced updates that must precede it. Uids are irrelevant for immediate updates, however, because the replicas constrain their ordering relative to other operations.

The specification of the complete service is given in Figure 2-3. As before we model the execution of a service as a sequence of events, one for each update and query operation performed by a front end on behalf of client. If *e* is an event in execution sequence E, G(e) denotes the set containing all events up to and including the most recent immediate update that precedes *e* in E. The second clause of the specification now requires that queries reflect the most recent immediate update in the execution as well as what the input label requires. The other clauses are unchanged except that the dependency relation for clause 4 is extended as follows:

dep(u, v) ≡ **if** immediate (u) **then** v ∈ P(u)
else if immediate (v) **then** u ∉ P(v)
else if forced(u) & forced(v) **then** v.uid < u.uid
else v.uid ∈ u.prev

Let q be a query. Then

1. q.newl ⊆ P(q).label.
2. q.prev ∪ G(q).label ⊆ q.newl.
3. u.uid ∈ q.newl ⇒ for all updates v s.t. dep(u, v), v.uid ∈ q.newl.
4. q.value = q.op (Val (q.newl)).

Figure 2-3: Specification of the Service.

2.2.1. Implementation of Forced Updates

We must provide a total order for forced updates and a way to relate them to causal updates and queries. This is accomplished as follows. As before, we represent uids and labels by multipart timestamps, but the timestamps have one additional field. Conceptually this field corresponds to an additional replica R that runs all forced updates; the order of the forced updates is the order assigned to them by this replica, and is reflected in R's part of the timestamp. Therefore it is trivial to determine the order of forced updates: if u and v are forced updates, u.uid < v.uid if $u.uid_R < v.uid_R$.

Of course, if only one replica could run forced updates, there would be an availability problem if that replica were inaccessible. To solve this problem, we allow different replicas to act as R over time. We do this by using a variant of the primary copy method [1, 28, 29] with view changes [6, 7] to mask failures. An active view always consists of a majority of replicas; one of the replicas in the view is the designated *primary* and the others are *backups*. The current primary is in charge of R's part of the timestamp as well as its own, and all forced updates are handled by it. Since we assume forced updates are relatively rare, it is unlikely that the primary will be a bottleneck.

To execute a forced update *u*, the primary carries out a two-phase protocol. In phase 1 it assigns a uid to the update by advancing R's part of the timestamp and merging it with *u.prev*. Then it creates a log record for *u* and sends this record to the backups in a message that contains in addition the log records for any earlier forced updates that have not yet committed. The update can commit as soon as a *sub-majority* of the backups acknowledge receipt of this message. (A sub-majority is one less than a majority of all the replicas in the service; once a sub-majority of backups know about the update, a majority knows (since the primary does too) and it is safe to commit the update since its effects will persist into subsequent views.) When the operation commits, the primary adds the record to its log, applies the update to the value if it is ready, and replies to the front end. The backups are informed about the commit in subsequent gossip messages.

A view change is accomplished by a two phase protocol conducted by a coordinator who notices a failure or recovery of a replica. The other replicas act as participants; the coordinator can go ahead with the view change if a sub-majority of the replicas agree to act as participants. The view change must ensure that all committed forced updates persist into the next view. In phase one of the view change, each participant informs the coordinator about the most recent forced update it knows. Note that any update that may have committed in the old view will be known to at least one member of the new view. When a sub-majority of replicas have responded, the coordinator sets R's part of the timestamp for the new view to the largest value it knows for any forced update. The primary of the new view will carry out the two-phase protocol for any uncommitted forced updates.

Forced updates do not interfere with the execution of causal updates or queries; all replicas proceed with these as before, including replicas that are disconnected from the current active view. Furthermore, a view change has no effect on what causal updates are known in the new view; instead, these continue to be propagated by gossip just as before.

If an application were configured to have only forced updates, our method would provide a standard primary copy implementation, with one exception: queries could be ordered relative to updates using timestamps, which allows them to be processed at backups, thus spreading system load. Queries would see the effects of updates that causally preceded them, but might not observe the effect of the most recent update. Such behavior is perfectly acceptable in many applications in which stale data is permitted; in fact our system provides a good way of controlling how stale the information is permitted to be. In addition, queries could be guaranteed to see recent updates by running them at the primary.

2.2.2. Implementation of Immediate Updates

To implement an immediate update *u* we need to carry out a global communication among all the replicas during which the system determines what updates precede *u* and computes the label *u.prev*, which identifies all such updates. At the end of this step *u* can actually be performed.

The primary of the active view will carry out immediate updates, but only if the view contains all replicas of the service. Timestamps for immediate updates are assigned in the same way as for forced updates, by using the R part of the timestamp. We use a three-phase algorithm [33] to perform an immediate update. Phase 1 is a "pre-prepare" phase in which the primary asks every backup to send its log and timestamp. Once a backup replies to this message, it stops responding to queries; it can continue to process causal updates, but it cannot reflect them in its *val*. (We discuss why these constraints are necessary below.) When the primary receives information from all the backups, it enters phase 2, the "prepare" phase; at this point it becomes unable to process queries and to reflect causal updates into its *val*. The primary processes the reply messages as gossip, assigns *u* a timestamp by advancing the R part of its timestamp, creates a log record for *u*, and sends the record to the backups. When a sub-majority of backups acknowledge receipt of this record, the primary commits the operation: it enters the record in its log, performs the update (it will be ready because the primary heard about all updates in *u.prev* in the responses in phase 1), and sends the reply to the front end. The other replicas find out about the commit in gossip; since they are unable to process queries until they know about the commit, the gossip is sent immediately.

If a view change occurs, the participants tell the coordinator everything they know about immediate updates. Any operation known to be prepared will survive into the new view, and the primary of the new view will carry out phase 2 again; such an operation must survive, since the old primary may have already committed it. An operation not known to be prepared will be aborted; such an operation cannot have committed in the old view, since a commit happens only after a sub-majority of backups enter the prepare phase, so at least one participant in the new view will know about the prepare.

Now we discuss why backups cannot respond to queries once they enter the pre-prepare phase and why the primary cannot respond to queries once it enters the prepare phase. (Causal updates cannot be reflected in *val* during these phases for the same reason.) Recall that once an immediate update happens, any query must reflect the effects of that operation. However, once a backup is in the pre-prepare phase, or the primary is in the prepare phase, it does not know the outcome of the operation. (The primary doesn’t know because a view change may have occurred.) Returning a value that does not reflect the update is wrong if the operation has already committed; returning a value that reflects the update is wrong if the operation aborts. Therefore, the only option is to delay execution of the query.

Immediate updates slow down queries. In addition, if a replica becomes disconnected from the others while in phase 1 or phase 2, it will be unable to process queries until it rejoins a new active view. This is analogous to what happens in other systems that support atomic operations: reading is not allowed in a minority partition, since if it were inconsistent data could be observed [6].

We chose a three-phase protocol for immediate updates because if a failure prevents the primary from

communicating with the other replicas, a new majority view will be able to decide whether to commit or abort the immediate update without waiting for the old primary to recover and meanwhile preventing the processing of other client requests. This is important because while an immediate update is running, queries are blocked. A two-phase protocol would require fewer messages, but an inopportune failure would prevent the entire system from processing queries.

3. Performance

3.1. Normal case operation

System performance during normal operation (i.e., in the absence of failures) depends on the size and number of messages, delay as perceived by clients, and the load at the replicas. Message size is not a problem if services are small. We expect services to contain only a few replicas (e.g., seven); this issue is discussed further in Section 4.

Figure 3-1 shows the number of messages required for carrying out different kinds of operations (the figure ignores batching at the front end). Here N is the number of replicas, K is the number of update/ack pairs in a gossip message, and M is the smallest integer greater than N/2. We are assuming a gossip scheme in which each replica gossips with all the others, but only about the updates that it processed and that the recipient may not know. We are also assuming that acks are piggybacked on subsequent messages so that separate ack messages are not needed; piggybacking acks may delay them but this is not a problem because it only affects how long cids are kept in *inval*. The second term for the causal operations expresses the cost of gossip. The second term for forced operations expresses the cost of phase 2 of the protocol, which is done by gossip; the third term for the immediate operations is similar. It is clear from the figure that queries and causal updates require few messages, forced updates require about the same number of messages as in primary copy schemes, and immediate updates are expensive.

operation	number of messages
query	2
causal	2 + (N-1)/K
forced	2M + (N-1)/K
immediate	2N + 2(M-1) + (N-1)/K

Figure 3-1: Number of messages for different kinds of operations.

Updates cause no delay to clients since they are asynchronous. A client cannot receive the answer to a query until one message round trip after making the call (although it may be able to do useful work in the interim if some form of non-blocking call is provided for it). In addition, the reply to a query may be delayed because:

1. Information about some updates it depends on has not yet arrived at the replica that processed it. This is unlikely in the absence of failures because the front end always communicates with the same replica and because gossip is frequent.
2. It follows a forced or immediate update whose execution is not complete.

3. An immediate update is in progress.

Delays for the latter two reasons will be rare if forced and immediate operations are rare.

Each replica must receive and process client requests, send and receive gossip, and execute every update. Since the sending and receiving of messages is expensive, the use of gossip to reduce the number of messages that servers handle decreases the load. For example, our scheme places less load on servers than schemes such as the ISIS multicast [4] in which every update causes a message to be received at every server.

To get a sense of how well lazy replication would perform in practice, we implemented a prototype causal operation service and compared its performance with an unreplicated prototype. We considered only causal operations because lazy replication is intended for applications where most operations are causal and therefore forced and immediate operations have little impact on overall system performance. Our measurements indicate that gossip is an effective technique; it enables a replica to handle more operations per second than it could if it needed to receive a separate message for each update.

The prototypes implement a simple location service with insertion and lookup operations. They are implemented in Argus [25] and run on a network of VAXStation 3200's connected by a 10 megabit-per-second ethernet. An Argus program is composed of modules called *guardians* that may reside on different nodes of a network. Each guardian contains local state information and provides operations called *handlers* that can be called by other guardians to observe or modify its state; it carries out each call in a separate thread and in addition can run some background threads. A computation in Argus runs as an atomic transaction; transactions are not needed in our system and add to the cost of using the service, but the additional cost is incurred equally in both the replicated and unreplicated prototypes.

The replicated service is implemented as a number of guardians, one for each replica. Each guardian provides handlers that can be called to do lookups, inserts, and acks; acks can also be piggybacked on lookup and insert calls (as an additional argument). Each replica has a background thread that sends and receives gossip messages. The gossip thread first processes all waiting gossip messages; then, if G milliseconds have passed since it last sent gossip, it sends a gossip message to each of the other replicas. Each gossip message is constructed and sent separately; we do not use a broadcast or multicast mechanism. A replica gossips only about records it created that the recipient may not know.

The unreplicated service is implemented as a single guardian that is similar to the one that implements a replica. This guardian needs to handle dropped, re-ordered and duplicated messages from front ends, so it provides an ack handler and allows acks to be piggybacked on inserts and lookups. An update is processed when it arrives (unless it is a duplicate), and it's uid entered in *inval*; update records are not kept in the log, which contains only ack records. When an ack arrives, its cid is removed from *inval* and an ack record enters the log where it remains until sufficient time has elapsed.

The replicas simulated the front end calls in the experiments. This allowed us to control the rate at which calls arrived at the server and the operation mix, i.e., the proportion of inserts and lookups in the experiment. We used a

uniform-arrival-rate distribution and generated an arrival sequence that approximated the required mix in each prefix. Operation calls are simulated by an Argus thread. Each call consists of two parts, the computation part and the communication part. The computation part includes the actual work of doing the operation, e.g., checking for a duplicate, adding a record to the log, etc. The communication part is a busy loop that simulates the communication overhead at the server node, namely the receipt of the operation message and delivery to the Argus guardian, decoding of the message to obtain the arguments, construction of the reply message, and moving the message from the guardian onto the network. The duration of the communication overhead was determined by measuring the cost of null calls; such calls incur a cost of 5.6 ms at the server node (and another 5.4 ms. at the client).

The experiments used a gossip rate G of 100 ms. Acks were piggybacked on inserts and lookups. Also, operations were always ready to run when they arrived. This assumption holds during normal operation (which is what is of interest here) because front ends communicate with their preferred replicas in this case and gossip is frequent.

Figure 3-2 shows the behavior of a single replica in a three-replica system in which all three replicas are processing the same mix of operations arriving at the same rate. Curves are given for different operation mixes (all queries, 1% updates, 10% updates, 50% updates, 100% updates). The horizontal axis shows the request arrival rate in operations per second; the vertical axis shows the mean response time for operations at that arrival rate and operation mix. The response time measures only the time spent at the replica's node which in addition to the call processing time includes the time spent in the replica queue waiting to be processed; the response time as seen by the client (for queries) is 5.4 milliseconds larger, since it includes the overhead at the client node. Note that the actual capacity of the service is three times what is shown in the figure, since each of the replicas is processing the load that is shown.

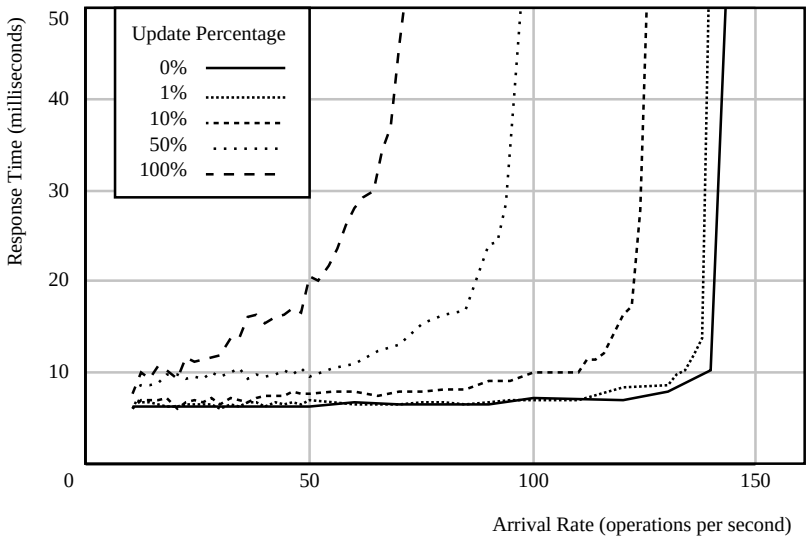

Figure 3-2: Capacity of a Single Replica. This figure shows the average response time as a function of operation arrival rate for selected operation mixes.

Figure 3-3 shows the results for the unreplicated system with the same operation mixes and arrival rates. By comparing the behavior of a replica with that of the unreplicated server, we can get a sense of the saving due to gossip. For example, in the operation mix of half updates and half queries, a replica saturates at approximately 90 operations per second, while the unreplicated system saturates at about l45 operations per second. However, a replica is actually processing 180 operations per second. (It is handling 45 queries, 45 updates from the client, and 90 updates that it receives in gossip from the other replicas.) The difference in the two cases is that the unreplicated server must process 290 messages per second, while the replica must process approximately 220 messages per second (180 messages for communicating with front ends, plus it sends 20 gossip messages and receives 20 gossip messages).

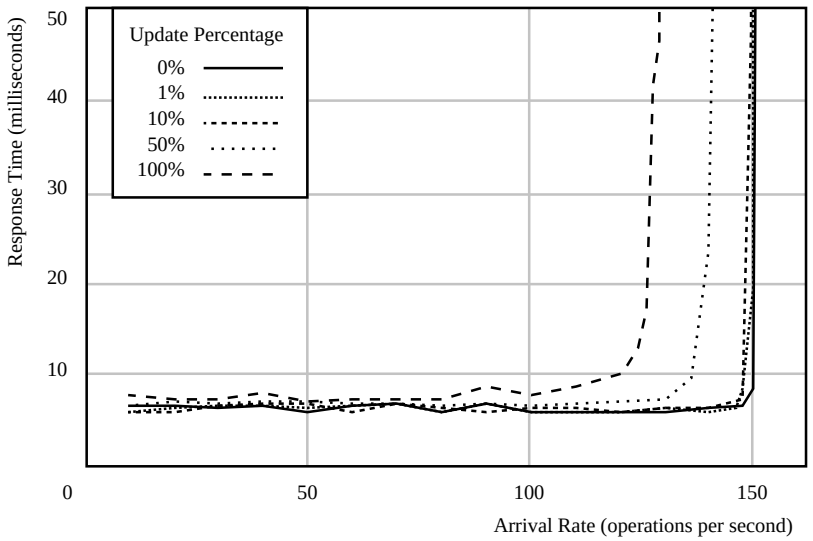

Figure 3-3: Capacity of the Unreplicated System.

The performance of the system is likely to be sensitive to the relative priorities of gossip and operations. The measurements above correspond to a system in which gossip has higher priority than operation processing: when there is a gossip message to send or receive, the gossip process will run. If operations have higher priority than gossip, this will probably lead to better response time, although gossip cannot be allowed to lag too far behind because it is important to propagate information about updates reasonably rapidly. We have not experimented with changes in relative priorities but it will be important to do this in a real implementation.

3.2. Reliability and Availability

The traditional way of achieving high reliability is to record information on a sufficient number of nonvolatile devices. For example, recording information on stable storage [21] insures that it will survive failure of its node and also a single media failure. However, this solution is wasteful, since we already provide a different kind of redundancy, namely recording updates at many replicas. Also it provides poor availability. If the one replica that records an update crashes or is partitioned from the other replicas before any of them know about the update, queries based on that update will be delayed.

A better approach to reliability takes advantage of replication. With this approach, an update is "stable" if it is recorded reliably at a sufficient number of replicas. For example, if information must survive a single failure, an update will be stable as soon as it is recorded at two replicas. A majority of replicas is not needed; instead the number of replicas is chosen based simply on the reliability requirements (e.g., we might use two replicas in a service containing seven of them). Recording the update on volatile storage will be adequate if these replicas are geographically distributed; if the replicas are close, the update can be recorded on volatile storage provided replicas have uninterruptible power supplies that allow them to copy volatile information to disk if there is a power failure. (This technique has been used to good effect in the Harp system [12].) A replica that loses its state in a crash recovers it by reading the state of enough other replicas so that it can determine the last update it processed.

Queries should not be permitted to observe the effects of an update until it is stable or else the following anomaly can arise: A query q observes update u which is not stable, and then u is lost (e.g., because the node that processed u and q failed before sending information about u to some other replica). A loss of causality like this should happen only in a catastrophe in which more failures occur simultaneously than the system is configured to handle.

Replicas can delay sending replies for updates to front ends until the updates are stable; if streaming is used between the front end and the replica, the replica also will not process a query that follows an update until the update is stable. Alternatively, the front end could send the update request to the appropriate number of replicas, and delay a query that follows the update until all have replied. In either case, the cost for causal updates increases. For example, in a system that survives one failure, a causal update now requires 4 + (N-2)/K messages, assuming that if the replica enforces the requirement, it notifies one other replica immediately (to reduce the delay until the update is stable), and if the front end does the enforcement, the gossip scheme avoids unnecessary propagation of information about the update. The scheme where the replica does the enforcement probably delays subsequent queries a little less assuming the front end is streaming queries to the replica.

Achieving reliability through replication has an impact on availability. Availability is an issue primarily for queries (since updates are asynchronous). If a query depends on updates that happened in the past, it is highly likely that it can be performed as long as one replica is up and accessible because gossip is frequent. However, a query that depends on a recent causal update might be a problem; since the query cannot be executed until the update is stable, its availability is similar to that of the update.

Forced and immediate updates are likely to be stable as soon as they are completed, since the number of replicas required for stability is probably less than a majority of replicas in the service. A query that follows a forced update probably requires a majority of replicas to be up and accessible, since this is needed to carry out the forced update. Immediate updates have an even larger effect on availability of queries, since queries are blocked while they are running. If a failure occurs while an immediate update is running, this has no effect on replicas that can join a new majority view; we chose a three-phase protocol for this reason. However, if a replica becomes disconnected from the others while running an immediate update, it cannot process any queries until the partition is repaired. These facts indicate that immediate updates must be rare if the system as a whole is to perform well. Note that if they are rare, the probability of a replica becoming disconnected while one is in progress is very small.

4. Scalability

In our scheme the number of replicas is independent of the number of clients. Since typically there will be large numbers of clients, and since as the size of a system increases, the number of servers can grow much more slowly than the number of clients, this is an important consideration. Having few replicas reduces the size of the timestamps, the storage requirements (since each replica needs to store the service state), and message traffic (since replicas need to communicate, even if gossip is done infrequently). Thus we expect the technique to work well in large systems. Below we discuss ways of controlling cost in large replicated services, and also what happens when there are many replicated services within a single system.

4.1. Large Services

Some applications may need a large number of replicas, either to provide adequate processing power, or to ensure that every client is "close" to a replica. When most calls to a service are queries, having lots of replicas can improve performance. However, having many replicas increases the cost of updates because more gossip messages must be sent and also because each update must be performed at every replica.

If many replicas are required, most of them can probably be read-only. Read-only replicas act as caches located at convenient locations in the network, e.g., one in each local area net. Such replicas only process queries; updates must still be sent to the regular read-write replicas. Having such replicas is effective provided a substantial portion of the system load is queries, or if fast processing of queries is the main performance issue. Timestamps need not contain entries for read-only replicas, so they will still be small. Read-only replicas must receive gossip to bring them up to date, but need not send gossip messages.

Sometimes the amount of gossip can be reduced by partitioning the service state. Partitioning can be used in any application in which different parts of the state are independent, i.e., each operation can be performed using the information in just one part of the state. For example, in a mail system there might be two partitions, the first storing mail for users with names in the first part of the alphabet, and the second storing the rest. A partitioned service still appears to be a single entity to clients. The replicas are divided into disjoint groups, each of which is responsible for a disjoint part of the state. All queries and updates concerning a particular part of the state are handled by the replicas in the group that manages that part and gossip is exchanged only among group members. The front end would maintain information about partitioning so that a client request could be sent to a replica in the request's group. The timestamps would contain components for all replicas; this is necessary to preserve causal order across the partitions. However, the timestamp components for replicas in other groups can be ignored when processing a query or update: an operation is ready to execute if the timestamp components for its group indicate that it is ready. Therefore, the extra components do not delay operation execution.

Sometimes clients of a service expose information about the service state only through calls on service operations. When this is true, front ends need not insert timestamps in client messages, since they are not needed to preserve causality, thus reducing the size of client messages. In addition, it is possible to use a hierarchical structure for the service. The idea is to partition the clients among a number of different replica groups, each consisting of a small number of replicas, and each having its own timestamps. Clients communicate only with replicas in their own

group; they use only that group's timestamps and never exchange timestamps with one another. The replica groups communicate with one another via a lower-level replica group, i.e., they are clients of the lower-level group; in fact, the scheme can be extended to an arbitrary number of levels. This scheme has been proposed for the garbage collection service [18]; in this application, a client's query is ordered only with respect to its own updates, although the speed with which inaccessible objects are discarded depends on how quickly global information propagates from one replica group to another (via the lower-level replica group). Another application that could profit from this approach is deadlock detection [8].

4.2. Multiple Services

We now consider a system containing many services. Causality can be preserved across a number of services by using joint front ends (one per client node) that manage all of them. The front ends would maintain labels for each replicated service. (The multipart timestamps for the different services can be distinguished by having each one identify its service.) All the labels would be sent in client messages and merged into a front end's labels when a client message is received. Also, a front end would send all the labels in each message to a replica. Replicas would use only their own timestamps to determine when operations are ready, but they would keep copies of all the labels, merge in the foreign ones, and send all the labels back in replies, at which point the front end would merge again. In this way we can preserve causality, both intra-service and inter-service, without clients having to be concerned with the details.

This automatic technique would be a problem if there were many services that were causally related, since in this case messages would contain lots of label timestamps. In our experience, however, most services that can be implemented with lazy replication are not causally related, so there is no problem in practice. Most replicated services are used by encapsulating code that hides their existence from the rest of the system. For example, the garbage collection service is used only by the heap managers at the client nodes, while the version deletion service is used only by the concurrency control subsystem; neither of these is related to one another, nor are they related to the mail system. Each such service can be provided with its own front ends (one per client node), distinct from those for other services, that manage just the timestamps for that service and intercept only the client messages sent by the encapsulating code.

5. Comparison with Other Work

Our work builds on numerous previous results, including general replication techniques such as voting [2, 10, 14] and primary copy [1, 28, 29], but is most closely related to approaches that provide high availability for applications where operations need not be ordered identically at all replicas. This section discusses this closely related work.

The idea of exploiting system semantics to enhance availability appears first in [9, 36]. In these systems, servers are coresident with clients and propagate information about updates by means of gossip. Causality is easily preserved for a single client, since it always communicates with the server at its node; preservation of inter-client causality is not discussed. Having a server at every client wastes space (since the server state must be stored at every client node) and also wastes network bandwidth (since every client node must be notified about every update).

In the Grapevine system [5] and its successor [20] service nodes are distinct from client nodes. Every client operation is performed at a single server, and updates propagate in the background to other replicas. Thus Grapevine solves the problems of wasted space and network bandwidth but it sacrifices causality. For example, a single client's request may go to different replicas (e.g., if the first replica used by that client fails); a later query by a client may fail to observe an earlier update made by that client. Also, different clients typically use different replicas, and a client may not observe the effects of an update it learned about in a message from another client.

Our earlier work [16, 22] separates servers from clients and supports a limited form of causality. Client requests are executed at just one server, and information about updates is propagated in gossip. Timestamps are used to force queries to observe specified updates but there is no way to order updates; instead updates must be commutative, so that they can be ordered in different ways at different replicas without affecting what clients observe via queries.

The idea of allowing the designer of an application to choose from a set of primitives of differing strengths appears first in the work on ISIS [3] and later in Psync [27]. Both systems provide multicast communication mechanisms that can be used to provide a replicated service. The Psync approach is limited to two kinds of operations, commutative and totally ordered, and only one operation can be of the more efficient commutative type. For example, in a mail system the read_mail operation could be commutative, but then send_mail and all other updates would need to be totally ordered.

ISIS provides three multicast primitives, CBCAST, ABCAST, and GBCAST, that support orderings roughly equivalent to those of the causal, forced, and immediate operation types. Its implementation is based on the notion of process groups that contain both clients and servers; intra-group communication uses reliable multicast primitives. Every client request is sent to all group members, resulting in substantial message traffic unless the entire system is located on a single local area net that supports broadcast. To ensure eventual delivery of requests, messages contain information about past history. In the earlier version of ISIS [3], messages contained descriptions of earlier requests; this resulted in very large messages and a garbage collection problem (recognizing when old requests could be discarded).

ISIS now uses a multipart timestamp scheme [4] similar to ours. Timestamps are used to order client operations but they have fields for clients as well as servers. Each operation (both queries and updates) is assigned a timestamp by the client node (by advancing the client part of the timestamp), and a server must know about all operations with smaller timestamps before it can process a new request. The client either multicasts the request to all replicas, or sends it to a single replica, which multicasts to the others; in either case many more messages are sent than in our system, and furthermore queries as well as updates need to be multicast. In addition, messages contain large amounts of timestamp information. If a group contains all clients and servers, timestamps are large. If there is a separate group per client (containing that client and all the servers), timestamp are smaller and there are fewer messages since requests from one client need not be sent to other clients, but messages are bigger since timestamps for all such groups must be sent in all messages to preserve causality.

Thus, our system sends fewer messages than ISIS, and requires less space for timestamp information in messages. In addition it performs better in the presence of failures and recoveries than ISIS. Both systems do view changes in

such situations (although we need not do view changes for a service in which all operations are causal). However, the ISIS view change prevents processing of new requests while it is in progress and involves flushing the service state to all replicas in the new view. By contrast, our replicas can continue to process causal updates and queries during a view change and little information needs to be flushed (just ordering information about outstanding forced and immediate updates). Furthermore, our system can tolerate network partitions, while ISIS cannot.

6. Conclusion

This paper has described a new replication method. The method supports three kinds of ordering for updates: causal, forced, and immediate operations. It can be used in many applications, including location services, distributed garbage collection, and mail systems. It performs better than alternative techniques such as reliable group multicast.

The method is generic and can be easily instantiated to provide a particular service. The instantiator provides a nonreplicated implementation of the application's operations and defines the ordering types of the updates. The user of the application just calls the operations. All details of replication and distribution are taken care of automatically.

The method is intended for applications in which most update operations are causal. In this case, it provides excellent performance. Client requests encounter low delay and the system has low overhead in terms of number and size of messages and overhead at the replicas. Our performance expectations are backed up by the experiments discussed in Section 3. A real implementation of a generic service that can be instantiated to provide replicated services is underway [17].

The forced and immediate operations are important because they increase the applicability of the approach, allowing it to be used for applications in which some updates require a stronger ordering than causality. Forced updates are also interesting in their own right. They can be used in an application that requires identical update orderings at all replicas with a cost comparable to techniques such as voting and primary copy. The method generalizes these approaches, however, because it gives queries access to stale data while ensuring that causality is preserved.

When confronted with the need for a highly available service, a designer has a limited number of choices if preservation of consistency is a goal. One possibility is to use standard atomic methods, in which all operations run in the same order at all replicas. However, some applications can tolerate having updates that run in parallel be executed in different orders at different replicas. In this case our method can be used. It reduces delay and number of messages in exchange for timestamp information in messages. Our method will be worthwhile provided this information remains reasonably small.

Acknowledgments

We wish to thank Boaz Ben-Zvi, Phil Bernstein, Andrew Black, Dorothy Curtis, Joel Emer, Bob Gruber, Maurice Herlihy, Paul Johnson, Elliot Kolodner, Murray Mazer, Bill Weihl, and the referees for their suggestions on how to

improve the paper.

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. Bernstein, P. A., Goodman, N. "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases". *ACM Transactions on database Systems 9*, 4 (December 1984), 596-615.

3. 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.

4. Birman, K. and Schiper, A. and Stephenson, P. "Lightweight Causal and Atomic Group Multicast". *ACM Transactions on Computer Systems 9*, 3 (August 1991).

5. Birrell, A., Levin, R., Needham, R., and Schroeder, M., "Grapevine: An Exercise in Distributed Computing". *Comm. of the ACM 25*, 4 (April 1982), 260-274.

6. El-Abbadi, A., and Toueg, S. Maintaining Availability in Partitioned Replicated Databases. Proc. of the Fifth Symposium on Principles of Database Systems, ACM, 1986, pp. 240-251.

7. El-Abbadi, A., Skeen, D., and Cristian, F. An Efficient Fault-tolerant Protocol for Replicated Data Management. Proc. of the Fourth Symposium on Principles of Database Systems, ACM, 1985, pp. 215-229.

8. Farrell, K. A. A Deadlock Detection Scheme for Argus. Senior Thesis. Laboratory of Computer Science, M.I.T., Cambridge, MA.

9. Fischer, M. J., and Michael, A. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. Proc. of the Symposium on Principles of Database Systems, ACM, March, 1982, pp. 70-75.

10. Gifford, D. K. Weighted Voting for Replicated Data. Proc. of the Seventh Symposium on Operating Systems Principles, ACM SIGOPS, Pacific Grove, CA, December, 1979, pp. 150-162.

11. Gifford, D. K. Weighted Voting for Replicated Data. Proc. of the Seventh Symposium on Operating Systems Principles, ACM, December, 1979, pp. 150-162.

12. Barbara Liskov and Sanjay Ghemawat and Robert Gruber and Paul Johnson and Liuba Shrira and Michael Williams. Replication in the {Harp
file system. SOSP13, , 1991, pp. 226--238.}

13. Heddaya, A. and Hsu, M. and Weihl, W. "Two Phase Gossip: Managing Distributed Event Histories". *Information Sciences: An International Journal 49*, 1-2 (Oct./Nov. 1989). Special issue on databases.

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

15. Hwang, D. 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.

16. Ladin, R., Liskov, B., and Shrira, L. "A Technique for Constructing Highly-Available Services". *Algorithmica 3* (1988), 393-420.

17. Ladin, R. and Mazer, M. S. and Wolman, A. A General Tool for Replicating Distributed Services. Proceedings of 1st International Conference on Parallel and Distribut ed Information Systems, December, 1991.

18. Ladin, R., *A Method for Constructing Highly Available Services and An Algorithm for Distributed Garbage Collection*. Ph.D. Th., M.I.T. Laboratory for Computer Science, Cambridge, MA, May 1989.

19. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System". *Comm. of the ACM 21*, 7 (July 1978), 558-565.

20. Lampson, B. W. Designing a Global Name Service. Proc. of the 5th Symposium on Principles of Distributed Computing, ACM SIGACT-SIGOPS, August, 1986, pp. 1-10.

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

22. Liskov, B., and Ladin, R. Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. Proc. of the 5th ACM Symposium on Principles of Distributed Computing, ACM, Calgary, Alberta, Canada, August, 1986.

23. Liskov, B., Scheifler, R., Walker, E., and Weihl, W. Orphan Detection (Extended Abstract). Proc. of the 17th International Symposium on Fault-Tolerant Computing, IEEE, Pittsburgh, Pa., July, 1987, pp. 2-7.

24. Liskov, B., Bloom, T., Gifford, D., Scheifler, R., and Weihl, W. Communication in the Mercury System. Proc. of the 21st Annual Hawaii Conference on System Sciences, IEEE, January, 1988, pp. 178-187.

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

26. Mills, D. L. Network Time Protocol (version1) specification and implementation. DARPA-Internet Report RFC-1059, DARPA, 1988.

27. Mishra, Sh., Peterson, L. L., Schlichting, R.D. Implementing Fault-Tolerant Replicated Objects Using Psync. Proc. of the Eighth Symposium on Reliable Distributed Systems, October, 1989.

28. Oki, B. M., and Liskov, B. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. Proc. of the 7th ACM Symposium on Principles of Distributed Computing, ACM, Aug., 1988.

29. Oki, B. M. Viewstamped Replication for Highly Available Distributed Systems. Technical Report MIT/LCS/TR-423, M.I.T. Laboratory for Computer Science, Cambridge, MA, August, 1988.

30. Parker, D. S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C. "Detection of Mutual Inconsistency in Distributed Systems". *IEEE Transactions on Software Engineering SE-9* (May 1983), 240-247.

31. Schmuck, F. B. The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems. Tech. Rept. 88-928, Dept. of Computer Science, Cornell University, Aug., 1988.

32. Schwarz, P., and Spector, A. "Synchronizing Shared Abstract Types". *ACM Trans. on Computer Systems 2*, 3 (August 1984).

33. Skeen, D., and Wright, D. D. Increasing Availability in Partitioned Database Systems. Technical Report 83-581, Dept. of Computer Science, Cornell University, Ithaca, N. Y., 1984.

34. Weihl, W. and B. Liskov. "Implementation of Resilient, Atomic Data Types". *ACM Transaction on Programming Languages and Systems 7*, 2 (April 1985).

35. Weihl, W. "Distributed Version Management for Read-only Actions". *IEEE Trans. on Software Engineering, Special Issue on Distributed Systems 13*, 1 (1987), 55-64.

36. Wuu, G. T. J., and Bernstein, A. J. Efficient Solutions to the Replicated Log and Dictionary Problems. Proc. of the Third Annual Symposium on Principles of Distributed Computing, ACM, August, 1984, pp. 233-242.

Table of Contents

1. Introduction 1
2. The Replicated Service 2
    2.1. Causal Operations 3
        2.1.1. Implementation Overview 5
        2.1.2. Processing at Each Replica 8
        2.1.3. Optimizations 10
        2.1.4. Analysis 11
    2.2. Other Operation Types 15
        2.2.1. Implementation of Forced Updates 15
        2.2.2. Implementation of Immediate Updates 17
3. Performance 18
    3.1. Normal case operation 18
    3.2. Reliability and Availability 21
4. Scalability 23
    4.1. Large Services 23
    4.2. Multiple Services 24
5. Comparison with Other Work 24
6. Conclusion 26

List of Figures

Figure 2-1: Specification of the Causal Operations Service. 4
Figure 2-2: The state of a replica. 6
Figure 2-3: Specification of the Service. 15
Figure 3-1: Number of messages for different kinds of operations. 18
Figure 3-2: Capacity of a Single Replica. This figure shows the average response time as a function of operation arrival rate for selected operation mixes. 20
Figure 3-3: Capacity of the Unreplicated System. 21