Lazy Replication: Exploiting the Semantics of Distributed Services

[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. In this paper, we propose lazy replication as a way to preserve consistency by exploiting the semantics of the service's operations to relax the constraints on ordering. Three kinds of operations are supported: operations for which the clients define the required order dynamically during the execution, operations for which the service defines the order, and operations that must be globally ordered with respect to both client ordered and service ordered operations. The method performs well in terms of response time, amount of stored state, number of messages, and availability. It is especially well suited to applications in which most operations require only the client-defined order.

[1]This research was supported in part by the National Science Foundation under Grant CCR-8716884 and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

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*. This paper describes a new method for constructing highly available services to be used in a distributed environment. Our method takes the semantics of the service into account to relax constraints on the implementation. Weakening constraints in this way can improve response time and increase availability. We take advantage of the weak constraints by having an operation call happen at a single replica. The effects of the call are then propagated to other replicas by *lazy* exchange of *gossip* messages — hence the name "lazy replication". We guarantee, however, that the replicated service appears to clients to provide the same behavior as a single copy.

A *service* provides operations that can be used by its clients to interact with it. There are two kinds of operations: *update* operations, which modify (but do not observe) the service state, and *query* operations, which observe (but do not modify) the state. (Operations that both modify and observe the state can also be provided.) *Clients* make use of the service by calling its operations. Our method allows clients to specify the order in which updates must be executed, and to specify what updates must precede a query so that their effects are recorded in the state observed by the query.

To illustrate how semantics can be used to relax constraints, consider an electronic mail system with *send_mail* and *read_mail* operations. 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. (Here, a client is the program acting on behalf of a person who is the real sender or receiver of the mail.) Therefore, the mail system can leave the execution order of these send_mail operations unrestricted, which can lead to better performance. However suppose client *c1* sends two messages to clients *c2* and *c3*, and the second message refers to information contained in the first one. Clearly, *c1* expects *c2* to read its messages in the order in which they were issued. Furthermore, if as a result of reading *c1*'s two messages, *c2* sends an inquiry message to *c3*, *c2* would expect its message to be delivered after *c1*'s messages.

With our method, clients indicate what updates must precede the execution of a call of a query operation (such as read_mail). We provide three ways for clients to control the order in which updates are executed. The first, the *client-specified order*, would be appropriate for send_mail operations. Here the client explicitly identifies all updates that must precede a particular call. The service performs updates in an order consistent with the client-specified order. Updates that are not ordered may be executed in different orders at different replicas, so this method is appropriate only when the execution order of unordered calls does not matter, as in the mail system example above.

We also support two stronger orders for updates, the *server-specified order* and the *global order*, which can be used when the client-specified order is insufficient to guarantee correct behavior. The system guarantees that server-ordered operations are performed in the same order at all replicas; the order for parallel server-ordered updates can be chosen non-deterministically, but once chosen it must be followed by all replicas. Such an order 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.

The relationship of server-ordered operations and client-ordered updates is not constrained by the service but instead is determined by the client. For example, a client might require that a send_mail operation be ordered after a particular add_user operation, or might choose to leave the order unspecified. Therefore usually there will be no global total order that relates all client-ordered and server-ordered updates.

Sometimes, however, a global total order is required and therefore we provide globally-ordered updates, which are guaranteed to be totally ordered with respect to all other operations, both updates and queries. Such an order would be useful for an operation that removes an individual from a classified mailing-list "at once", i.e., no messages addressed to that list should be delivered to that user after the remove operation returns. Such an "immediate" operation will behave in a way that is consistent with external events [14].

In this paper we describe a replication method that allows construction of highly-available services. The service operations are individually atomic; our scheme does not support multi-operation transactions. The method has the following desirable properties:

1. It allows three categories of update operations within a service: client-ordered operations (e.g., send_mail), server-ordered operations (e.g., add_user), and globally-ordered operations (e.g., delete_user from a mailing-list).
2. It performs 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 updates are client-ordered.
3. A user can define a replicated service without having to understand the complications due to distribution and replication. He or she simply instantiates our method to provide a particular service by supplying the definition of the replica data structures, their initial values, and the interpretation of the operations. In addition, the categories of the updates are defined when the service is instantiated.

We have applied the method to a number of applications, including detecting orphans in a distributed system [22], locating movable objects in a distributed system [15], garbage collection of a distributed heap [17, 21], deletion of unused versions in a hybrid concurrency control scheme [33], and deadlock detection in a distributed transaction system [9].

We begin in Section 2 by describing our assumptions about the environment. Section 3 describes the method that preserves the client-defined order; this is the most important part of our method and therefore it is presented in the most detail. The section also describes a prototype implementation of our technique, and gives some performance results; the data indicate that our replicated system has a larger operation processing capacity then a nonreplicated system with comparable response time. Section 4 extends our method to handle the other two orders. Section 5 discusses some issues that arise when using our method in practice. Section 6 relates our work to other replication methods, including the ISIS system [4, 3, 5], which is the closest to our approach. We conclude with a discussion of what we have accomplished.

2. The Environment

The method is intended to be used in an environment in which individual computers, or *nodes*, are connected by a communication network. The network has an arbitrary topology; for example, it might consist of a number of local area nets connected via gateways to a long-haul network. Both the nodes and the network may fail, but we assume the failures are not Byzantine. The nodes are failstop processors [31]. The network can partition, and messages can be lost, delayed, duplicated, and delivered out of order. The configuration of the system can change, that is, nodes can leave and join the network at any time.

We assume that nodes have loosely synchronized clocks that never run backwards. The correctness of our protocol depends only on the monotonicity assumption, but performance can suffer when clocks drift too far apart. There are protocols that with low cost synchronize clocks in geographically distributed networks, e.g., the NTP protocol [24] provides a clock skew on the order of a hundred milliseconds. 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 [20] periodically and recovered after a crash.

3. The Client-Order Method

In this section we describe the service that supports the client-ordered operations. We begin by giving an overview of what a service provides to clients, and then provide a formal specification for the service. Next we present the overall architecture of a service, describe the mechanisms and protocols used in our implementation, and prove that our implementation is correct. Finally, we discuss the performance of our method.

3.1. Client Interface

For the client-specified order, clients identify the updates that must precede a particular update or query. This is accomplished as follows: Every invocation of an update operation returns a unique identifier, *uid*, that names that invocation. In addition, every (query and update) operation *o* takes a set of uids as an argument; we will refer to such a set as a *label*. The label identifies the updates whose execution must precede the execution of *o*. In the mail service, for example, the client can indicate that one send_mail must precede another by including the first's uid 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.

Thus, the service operations are:

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). We require that clients not create uids but only use ones returned by the service. Given this constraint, the service will be able to execute operations in an order consistent with that indicated by their labels.

A label may include the uids of both the client's own update operations and those executed by other clients that it might have observed. The client learns about operations of other clients either directly, by communicating with other clients, or indirectly, by viewing the service's state. Clients must send labels in messages they exchange so that they can provide the proper ordering information to the service: any message from one client to another that reveals information about the state of the service should include a label that corresponds to the updates reflected in that state.

Since the service respects only explicitly specified dependencies, a client must make sure all its intended ordering requirements are conveyed to the service in the argument label. One way to manage labels is to have the system append them to all messages automatically and merge them whenever messages are received. Using this approach will lead to the well known causal or "happened before" order defined by Lamport [18]. However, better performance can be achieved if clients control the use of labels explicitly. This "explicit" approach can lead to smaller messages, since often it will not be necessary to include labels in them; this point is discussed further in Section 5.2. Also, the client has considerable freedom in how it chooses the label sent to the service. For example, if the client knows about some update *u*, but wants to run an operation that need not be after *u*, it does not include *u*'s uid in the label of the call. Letting operations "run in the past" like this can improve performance as discussed further below.

3.2. Service Specification

We model an execution of a service as a sequence of events, one event for each update and query operation performed by 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, we denote by P(e) 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 of the client-order service is given in Figure 3-1. Since the client is able to observe the state resulting from execution of updates only by executing queries, the specification relates the updates and their dependency constraints to what can be observed by queries.

Let q be a query. Then

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

Figure 3-1: Specification of the Client-Order Service.

The first part of the specification states that the label returned to the client identifies all updates the client required plus possibly some additional ones. The second part 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 of the constraint on clients not to create uids.

The third part of the specification defines the relationship between the value and the label returned by a query: the result returned must be computed in a state Val arrived at by performing the updates identified by the label in an order consistent with the dependency relation. For label L,

$$\text{Val(L)} = \text{compute (init, } \{ \text{ u} \mid \text{u.uid} \in \text{ L } \})$$

where "init" is the initial state of the service, and *compute* performs the updates identified by L in an order consistent with the dependency order:

$$\text{dep (u, v)} \Rightarrow \text{v.op is executed before u.op.}$$

Note that operations not ordered by clients can be performed in arbitrary order in computing Val. Note also that this clause guarantees that if the returned label is used later as input to another query, the client will observe the effects of all updates it has already observed. Finally, the fourth clause of the specification requires that all updates identified by the label have actually occurred.

3.3. Service Architecture

A service is implemented by a number of *replicas*. We assume there is a fixed number of replicas residing at fixed locations and that clients 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, and that the state of each replica is kept on stable storage [20] and restored after a crash; this assumption is discussed in Section 3.6.2.

Each client runs a *front end* at its node. When the client calls a service operation, its front end sends an appropriate request message to some convenient replica. The replica executes the operation and sends back a reply message. Replicas communicate new information (e.g., about updates) among themselves by lazily exchanging *gossip* messages.

If the response is slow, the front end might send the request to a different replica or it might send the same request to several replicas in parallel. Therefore a single operation can result in duplicate messages being sent to several replicas. In addition, messages may be duplicated by the network. Our scheme can cope with messages that are lost, delayed, duplicated, and delivered out of order; it hides the fact that a call has resulted in multiple messages from clients, and guarantees that calls are performed at most once at each replica. To allow the service to distinguish between different calls with the same parameters and thus to prevent multiple executions of an 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.

Interdependent operations may execute at distinct replicas; for example, a read_mail operation that is supposed to observe the effects of a send_mail operation may be sent to a replica that does not yet know about the send_mail. Replicas use labels to determine whether all updates on which an operation depends are known locally. A query operation will be delayed until all needed updates are known; this is why it is advantageous for clients to run queries "in the past", since doing so increases the probability that all needed updates are known when the query arrives at a replica. In the case of an update, there is no delay in responding to the client; it is sufficient to assign the update a uid and record it for later execution.

3.4. Implementation

In this section we describe the implementation of the service in detail. We describe the implementation of uids and labels, present the replica data structures, explain how the implementation controls the size of the data structures and the volume of communication while guaranteeing that updates are executed only once, and present the detailed replica protocol.

For our method to be efficient, we need to represent labels compactly and we need 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 of size *n*

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

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 our own earlier work [21, 16].)

We implement both uids and labels as 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*. We implement the dependency relation 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 the label represented by *t* identifies a subset of the updates identified by *t'*. This test is used to determine whether an operation is ready to be executed.

```
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: {log-record}    % 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 3-2: The state of a replica.

Figure 3-2 shows 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.) The *log* is a set of timestamped records that correspond to the messages that the replica received directly from the clients and those that were processed at other replicas and have been propagated to it in gossip. There are two kinds of client messages, updates and acks; acks are discussed below. The replica's timestamp, *rep_ts*, expresses the extent of the replica's knowledge of messages that were either directed to it or that it heard about via gossip; it uniquely identifies the set of records in the node's log. A replica increments its part of its timestamp when it processes a client message. Therefore, the value of the replica's own part of its timestamp *rep_ts* corresponds to the number of messages that it has accepted directly from clients. The replica increments other parts of its timestamp when it receives gossip from other replicas containing messages it has not seen yet. The value of any other part *i* of *rep_ts* counts the number of client messages processed at replica *i* that have propagated to this replica via gossip.

Each replica maintains in *val* its view of the current state of the service. It obtains this view by executing the updates in the log in an order consistent with the dependency order. The label timestamp *val_ts* is the merge of the timestamps of updates reflected in *val*. It is used to determine whether an update or query is ready to execute; an operations *op* is ready if $op.prev \leq val\_ts$. The set *inval* is used to prevent executing duplicate updates. We store an update record in *inval* the first time an update is executed, and use the cid to avoid executing that update again. Since the same update may be assigned more than one uid timestamp (this may happen when the front end sends the update to several replicas), the timestamps of duplicate records for an update are merged into *val_ts*. In this way we can honor the dependency relation no matter which of the duplicates the client knows about (and therefore includes its uid in future labels).

The description above has 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 clients 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* client records created by replica *j*. Every record *r* contains the identity of the replica that created it in field *r.node*. The predicate *isknown(r)* holds at replica *i* if *i* knows that *r* has been received by every replica:

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

The second implementation issue is controlling the size of *inval*. It is safe to discard a record r from *inval* only if we can be certain that the replica will never attempt to apply *r's* update to *val* in the future. To achieve such a guarantee, we need to establish an upper bound on when messages containing information about *r's* update can arrive at the replica.

A front end will keep sending messages for an update until it receives a reply. The 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 client calls. Acks are added to the log when they arrive at a replica and are propagated to other replicas in gossip.

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, and we require that the time in an ack be greater than or equal to the time in any of the messages for the ack's update. We discard an update message m because it is "late" if

$m.time + \delta <$ the time of the replica's clock

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

$a.time + \delta <$ 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 update record r from *inval* as soon as an ack for *r's* update is in the log and all records for *r's* update have been discarded from the log. The former condition guarantees a duplicate of *r's* update will not be accepted from the client or network; the latter guarantees a duplicate will not be accepted from gossip (see Section 3.5 for a proof). A replica can discard ack a from the log once a is known everywhere, *a's* update has been discarded from *inval*, and all client-introduced duplicates of *a's* update are guaranteed to be late at the replica. Note that we rely on the clock monotonicity assumption here.

For the system to run efficiently clocks of servers and clients 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's clock is slow, its messages may be discarded even though it just sent them. The delay δ must be chosen to accommodate both the anticipated network delay and the clock skew.

3.4.1. 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.

Processing an update message:

Replica i discards an update message u from a client if it is late (i.e., if $u.time + \delta <$ the time of the replica's clock) or it is a duplicate (i.e., a record r such that $r.cid = u.cid$ exists in *inval* or 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* }

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 of updates (and other messages) can be sped up by maintaining both the log and *inval* as hash tables hashed on the cid.

Processing a query message:

When replica *i* receives a query message *q*, it needs to find out whether it has all the information required by the query's input label, *q.prev*. Since *val_ts* identifies all the updates that are reflected in *val*, the replica compares *q.prev* with *val_ts*. 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. The two timestamps *q.prev* and *val_ts* 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 client.

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 proceeds as follows: If $m.ts \leq j$'s timestamp in *ts_table*, i discards the message since it is old. 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:

   $rep\_ts := \text{merge}(rep\_ts, m.ts)$

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. $\exists$ no $r' \in comp$ s.t. $r'.ts \leq r.prev$
       $comp := comp - \{ r \}$
       if r is not a duplicate of an element of *inval* **then**
           $val := \text{apply}(val, r.op)$
           $inval := inval \cup \{ r \}$
       $val\_ts := \text{merge}(val\_ts, r.ts)$

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:

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

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*:

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

The new value can be computed faster by first sorting *comp* such that record r is earlier than record s if $r.ts \leq s.prev$. Also, it is not necessary to send the entire log in gossip. Instead the sender can omit any records that it knows the receiver knows.

Since the decision to delete records from the log uses information from all other replicas, we may have a problem during a network partition. For example, suppose a partition divided the network into sides A and B and that 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 was pointed out by Wuu and Bernstein [34], 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.

3.5. Analysis

In this section we argue informally that our implementation is correct and makes progress, and that records are removed from the log and *inval* eventually. We also discuss an aspect of our implementation related to the availability of the server.

The specification in Figure 3-1 defines a centralized server in which each update is performed only once and is assigned a single uid. However, our 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, our 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 $r.ts \le t$.

We now consider the four clauses of the specification. The first clause requires that the updates identified by the query input label *q.prev* also be identified by the query output label *q.newl*. This clause follows immediately from the timestamp implementation of labels and from the query code, which returns only when $q.prev \le val\_ts$.

The second clause requires that the label *q.newl* be dependency complete. This clause 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 $r.ts \le u.prev$ and therefore the set of updates identified by a label timestamp is trivially dependency complete.

The third clause of the specification ties together the label *q.newl* and the value *q.value* returned by the query. It 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. Before proceeding with the proof of this clause, we establish several useful facts about our implementation. Each fact is stated as a lemma that refers to the state variables of a single replica. We assume in the proofs that each operation is performed atomically at a replica and also 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 client 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 client 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 client 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 has entered *inval*, or r 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 client. ☐

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 client 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 client 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 of the constraint that clients 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 a duplicate of r 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*, $r.prev \le rep\_ts$. Since *r.ts* differs from *r.prev* only in field j, where r was processed at j, and since r is known locally, $r.ts_j \le rep\_ts_j$ by Lemma 3. ☐

Lemma 8: For any update u, if there exists a record r for u s.t. $r.ts \le rep\_ts$, 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. $r.ts \le rep\_ts$ after that step. We need to show that u is reflected in the value. First, consider a gossip processing step. Since $r.ts \le rep\_ts$, we know $u.prev \le rep\_ts$. If r is in the log, it enters *comp* and either u is executed now or a duplicate record for u is present in the set *inval* and therefore u is already reflected in the value. If r is not in the log, then $r.ts \le rep\_ts$ 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 $r.ts \le rep\_ts$ before this step, our claim holds by the induction assumption. Otherwise, we have $\neg(r.ts \le rep\_ts)$ before this step and $r.ts \le rep\_ts$ 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 $u.prev \le rep\_ts$ before this step occurred. Therefore, any v that u depends on has already been reflected in the value by the induction assumption, and $u.prev \le val\_ts$. Therefore u is reflected in the value in this step. ☐

We are now ready to prove the third 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 $r.ts \le val\_ts$, *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 record r that entered the set *inval* when u was executed must be deleted from *inval* before u can be executed again. However, when r is deleted from *inval* no duplicate record for u is present in the log and an ack for r is present in the log. By Lemma 1, the presence of the ack guarantees that no future duplicate from the client or the network will reenter the log. Furthermore, when r is deleted from *inval*, *isknown(r)* holds, so by Lemma 4, all duplicates d of r have arrived at the replica. By Lemma 5, $d.ts \le rep\_ts$ 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 $r.ts \le val\_ts$ 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 $u.prev \ge s.ts$. 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 $u.prev \le val\_ts$; by Lemma 7, $u.prev \le rep\_ts$, 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 $u.prev \le rep\_ts$ 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 $u.prev \geq s.ts$, or there is already a duplicate for *v* 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.

The fourth clause of the specification requires that only updates requested by clients are executed by the service. It follows trivially from the code of the protocol, which only creates update timestamps in response to update messages from clients.

We have shown that our implementation is correct, i.e., that when queries return, their results satisfy the service specification. To ensure system progress, we now 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.

Next, we prove that records are garbage collected from the service state. 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 records are deleted from *inval*. To show that records are deleted from *inval*, we need the following lemma, which guarantees that an ack stays in the log long enough to prevent its update from reappearing in *inval*:

Lemma 9: *Isknown(a)* $\wedge$ type (*a*) = ack $\Rightarrow d.ts \leq rep\_ts$ 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 update record is deleted from *inval*. By Lemma 1, no duplicates of the update message from the client 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 records will be removed from *inval* assuming crashes and partitions are repaired eventually and clients send acks with big enough times.

Finally, we discuss how our implementation of uids and labels affects the availability of queries in our system. 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, our 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 in our implementation never needs to delay a query waiting for such an "extra" update to arrive because our 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.

3.6. Performance

In this section we discuss the performance of our client-order method. First we discuss the impact of replication on the rate at which queries and updates can be processed in the absence of failures. Then we discuss service availability and reliability.

3.6.1. Capacity

To determine how well our replication method performs in the absense of failures, we built a prototype implementation and compared its performance with an unreplicated prototype. To collect the performance measurements, we used a three replica system running on a network of VAXStation 3200's connected by a 10 megabits per second ethernet. We focused on measuring system *capacity*, which is the number of operations the system can process per unit time with a given response time. Our measurements indicate that the replicated system has a higher overall capacity then the nonreplicated system without significant degradation in response time.

Measurements of system capacity provide a basis for determining how many clients a service can support and still provide reasonable response time. To estimate the number of clients, the system designer needs to know what mix of operations occurs in the application and also how often individual clients use the service. For example, in a location service [15] that allows movable objects to be located in a network, updates are probably much rarer than queries since objects move infrequently. Such a system may have a mix in which 99% of the operation calls are queries; furthermore calls to the service are very infrequent, since clients cache recent information. A mail system may have a mix that is dominated by updates, since mail tends to be read in batches, but clients interact with the system relatively infrequently.

Our prototypes implement a simple location service with insertion and lookup operations. The prototypes were implemented in Argus [23]. An Argus program is composed of a number 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 [23]; transactions are not needed in our implementation 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. In addition, acks can 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 messages; then, if sufficient time has passed since it last sent gossip, it sends two gossip messages. Each gossip message is constructed and sent separately; we do not use a broadcast mechanism. A replica does not send its entire log in gossip messages; instead it sends only those records that it created[2] and the recepient may not know. Therefore, when a gossip message is received, we do not merge its entire timestamp into *rep_ts*; instead the sender's part of the timestamp in the message replaces its part of *rep_ts*.

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

To carry out our experiments, we had the replicas simulate the client calls. This allowed us to control the rate at which client calls arrived at the server and the operation mix, i.e., the proportion of inserts and lookups in the experiment. We use a uniform arrival rate distribution. To control the gossip rate we had the replicas send gossip using the following policy: gossip will be sent the next time the gossip thread runs provided it is at least G milliseconds since the last time gossip was sent.

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).

Figure 3-3 shows the results of experiments using a gossip rate G of 100 ms. In these experiments there were no separate ack messages; instead acks were piggybacked on inserts and lookups. Also, operations were always ready to run when they arrived. The figure shows the behavior of a single replica in a 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; the response time as seen by the client is 5.4 milliseconds larger, since it includes the overhead at the client node.

Figure 3-4 shows the results for the unreplicated system with the same operation mixes and arrival rates. From these two figures we can compare the capacities of the two systems. If the system is query-bound, the replicated system can handle nearly three times the number of operations without a degradation in response time (it has three times the capacity because each replica, as shown in Figure 3-3, handles almost as many operations per second as the unreplicated system). If the system is update-bound, the replicated system can still handle more operations than

---

[2]When a replica failure or recovery is detected, there is a period during which replicas send their entire logs in gossip messages.

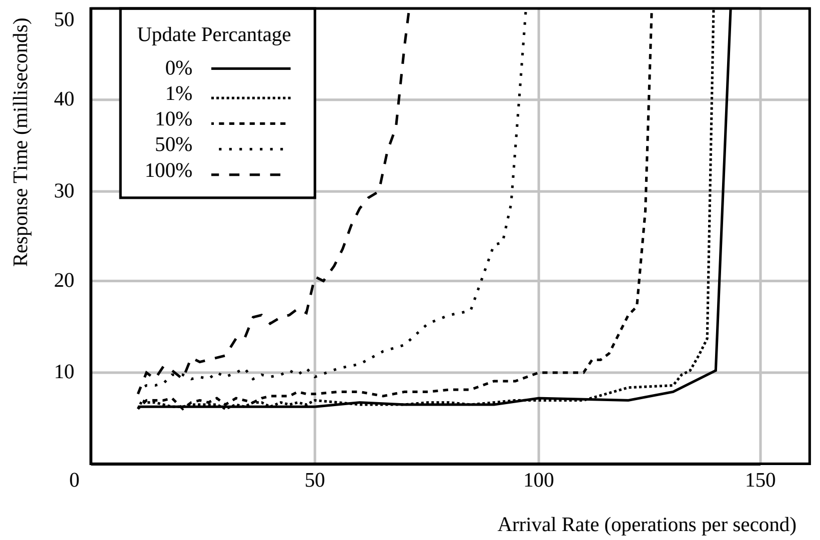

Figure 3-3: Capacity of a Single Replica. This figure shows the average response time as a function of operation arival rate for selected operation mixes.

the unreplicated one with some degradation of response time, in spite of the fact that ultimately each replica must perform each update. For example, if we are willing to accept a response time of 20 ms. for an operation mix of half queries and half updates, the capacity of the replicated system is 87 X 3 operations per second whereas the unreplicated system saturates at 140 operations per second. The reason for the improved capacity is that gossip messages typically contain entries for several updates, thus amortizing the communication overhead for the updates in gossip messages.

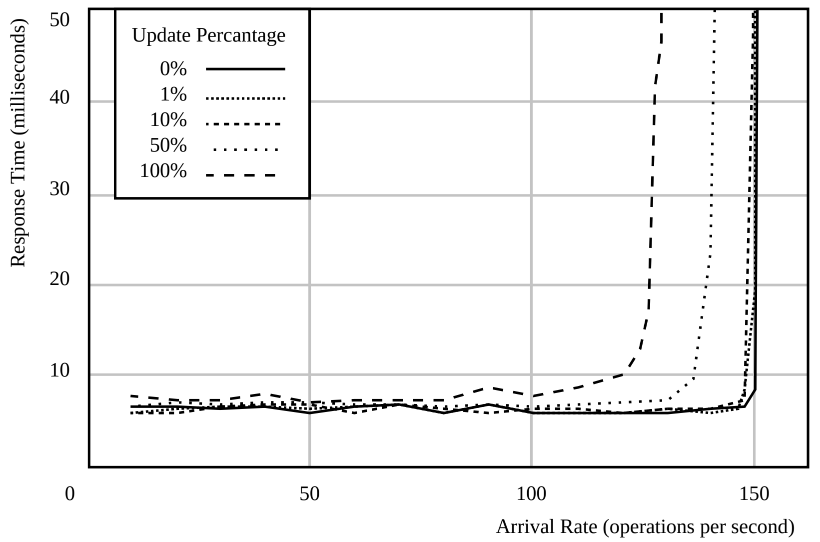

Figure 3-4: Capacity of the Unreplicated System.

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.

3.6.2. Availability and Reliability

In this section we discuss service availability (i.e., the probability that the service will be accessible when needed) and service reliability (i.e., the probability that the service will retain information entrusted to it). Our goal is to achieve better availability than a single copy system, and to provide the same kind of reliability that could be achieved without replication.

To understand the availability of our system, we look first at partitions and then at crashes. If a partition isolates a client from all replicas, no replication scheme will help. A more common case, however, is a partition that divides nodes into groups containing both replicas and clients. Our updates can proceed as long as the client's group contains a single replica. A partition could prevent queries from being processed if the query required information about an update that was processed on the other side of the partition. Since we assume that a partition divides the network into disjoint sets of nodes (client and service), there are two cases to consider:

1. A client requires information about an update it did earlier, but the replica that processed that update is separated from the client by a partition that happened after it sent the response to the client but before it communicated with other replicas. This situation is highly unlikely if gossip is frequent. We could make it even more unlikely by sending gossip in parallel with, or even before, sending the update response to the client.

2. A client requires information about an update done by a different client on the other side of the partition. This situation is impossible if the update was done after the partition formed, because there is no way for the clients to communicate the timestamp of the update (assuming no out-of-band communication). Therefore, the situation is similar to case 1.

Note that in either case we have better availability than replication techniques (e.g., [11], [26]) based on majorities.

Now we consider the case of crashes. Our scheme continues to provide service as long as a single site is up. However, we do have a problem with crashes that is similar to partition case (1), i.e., the crash happened just after the replica processed an update and responded to the client but before it communicated with other replicas. We can make this situation less likely by sending gossip before or in parallel with the update response. If the probability of failure is still considered too high, we can do the following: a replica notifies some number of other replicas, and waits for acknowledgments, before sending the client response. Note that this solution differs from voting [11] and primary copy schemes [1, 26] in that a majority of replicas is not needed. For example, only two replicas out of five or seven might be involved in an update. However, the extra communication does mean that the availability of updates will decline and response time will increase.

Finally we consider the reliability of the service. If replicas process updates without communicating with one another, stable storage seems wise, since otherwise the crash of a single replica could lose some updates forever. In particular, a replica would need to log an update on stable storage before responding to the client. However, if replicas communicate (with acknowledgments) before replying, stable storage is not needed. Storing the new update at more than one replica protects it against a certain number of failures (e.g., if two replicas have the information, the information will survive a crash of one of them). We still need to be concerned about a simultaneous failure of all replicas that store the information. The most likely cause of a simultaneous failure, a power failure affecting nodes that are physically close together, can be handled by equipping each replica with an uninterruptible power supply and a disk; in the case of a power failure, the replica's volatile information would be written to disk before it shuts down.

4. Other Orders

In the introduction, we described two additional update orderings that might be needed in applications. There are the server-ordered updates, such as the add_name operation in the mail system, which are totally ordered with respect to one another even when no dependency relationship is defined by the client. In addition, there are the globally-ordered updates, such as an operation to revoke user rights immediately, which are totally ordered with respect to all other operations.

In this section we discuss how our system can be extended to support server-ordered and globally-ordered updates. These two stronger orders are provided only for updates; queries continue to be ordered as specified by the client. We assume that the system knows *a priori* what the type of an update is; this would be established when the system is created or instantiated. Typically, a system will have several different update operations, e.g., send_mail, add_user. Each such operation will have a declared ordering type.

Like client-ordered updates, server-ordered updates take an input label and return a uid. The uids for server-ordered updates are totally ordered. Labels now identify both the client-ordered and server-ordered updates, and the input label for an update or query identifies the client-ordered and server-ordered updates that must precede it.

Unlike other operations, a globally-ordered update *u* does not take a label as an argument; instead, the system decides what operations precede *u*. Furthermore, although the system does assign *u* a uid, it does not return it to the client. Returning the uid is not necessary because any operation that runs after *u* returns is guaranteed to be ordered after *u*.

Let q be a query. Then

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

Figure 4-1: Specification of the Service.

The specification of the complete service is given in Figure 4-1. As before we model the execution of a service as a sequence of events, one for each update and query operation performed by a client. If *e* is an event in execution sequence E, we denote by G(e) the set containing all events up to and including the most recent globally-ordered update that precedes *e* in E.

The first clause of the specification now requires that queries reflect the most recent globally-ordered update in the execution as well as what the client required. Clauses 2 - 4 are unchanged except that the dependency relation for clause 3 is extended as follows:

$$\begin{aligned} \text{dep}(u, v) \equiv\ & \textbf{if } \text{globally-ordered}(u) \textbf{ then } v \in P(u) \\ & \quad \textbf{else } (\ v.\text{uid} \in u.\text{prev}\ ) \vee \\ & \quad\quad (\ \text{server-ordered}(u)\ \&\ \text{server-ordered}(v)\ \&\ v.\text{uid} < u.\text{uid}\ ) \end{aligned}$$

4.1. Implementation of Server-ordered Updates

To implement server-ordered updates we must provide a total order for them and we must provide a way to relate them to client-ordered updates and queries. This is accomplished as follows. As before, we represent update-ids and labels by multipart timestamps, but the timestamps have one additional field. Conceptually this field corresponds to an additional replica R that runs all server-ordered updates; the order of the server-ordered 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 server-ordered updates: if u and v are server-ordered updates, u.uid < v.uid if $u.uid_R < v.uid_R$.

Of course, if only one replica could run server-ordered updates, we would have 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 the primary copy method [1, 27, 26] with view changes [8, 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 server-ordered updates are handled by it.

To carry out a server-ordered 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 it to the backups. The operation can commit as soon as:

1. a *sub-majority* of the backups acknowledge receipt of its record, and
2. all earlier server-ordered updates have committed.

(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, then a majority knows (since the primary does too), and therefore it is safe to commit the update since its effects will persist into subsequent views.) When the operation commits, the primary adds its record to its log, applies the update to the value if it is ready, and responds to the client. 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 server-ordered updates persist into the next view. In phase one of the view change, each participant informs the coordinator about all server-ordered updates 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. The coordinator aborts any server-ordered update *u* that depends on a server-ordered update not known to any member of the new view, i.e., it aborts *u* if it has no record of a *v* with $v.uid_R < u.uid_R$. (The update *u* must be uncommitted in this case because *v* is uncommitted and updates are committed in order.) Then it sets R's part of the timestamp for the new view to the largest value it knows for any (unaborted) server-ordered update. The primary of the new view will carry out the two-phase protocol for any remaining uncommitted updates.

In a system of N replicas, and when there are no failures, a server-ordered update requires 2M messages, where M is the smallest integer greater than N/2, and encounters a delay of roughly 2 message round trips. Its execution does not interfere with the execution of client-ordered 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 client-ordered updates are known in the new view. Instead, these continue to be propagated by gossip just like they were in the system of Section 3.

If there are many server-ordered updates, the primary may become a bottleneck. In this case, a voting method [11, 2, 13] could be used, but this has the disadvantage that it will be blocking (i.e., if the client becomes disconnected from the service during phase 1, no more server-ordered updates could be performed until the client recovered). Blocking can be avoided by using a three-phase protocol [32], but then more messages are needed than with our scheme. Another possibility is to have several different orders for the server-ordered operations. For example, in a mail system we might choose to have one order for add_user operations that add users whose names start with "a" through "m", and a different order for users whose names start with "n" through "z". Each order would have a different part of the timestamp, and would be managed by a different primary. However, it is not practical to have a large number of different orders, since this would lead to large timestamps.

4.2. Implementation of Globally-ordered Updates

To implement a globally-ordered 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.

Our implementation works as follows. We use the primary of the active view to carry out globally-ordered updates, but the primary will execute a globally-ordered call from the client only if the view contains all replicas of the service. We assign timestamps for globally-ordered updates in the same way as for server-ordered updates, by using the R part of the timestamp. We use a non-blocking three-phase algorithm [32]: if a failure causes the primary to be unable to communicate with the other replicas, a new majority view will be able to decide whether to commit or abort the globally-ordered update without waiting for the old primary to recover.

Phase 1 is a "pre-prepare" phase in which the primary asks every backup to send its log and timestamp. Once a backup receives this message, it stops responding to queries; it can continue to process client-ordered 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 client-ordered updates into its *val*. The primary computes *u.prev*, assigns *u* a timestamp by advancing the R part of the 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 client. 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 globally-ordered 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. (Client-ordered updates cannot be reflected in *val* during these phases for the same reason.) Recall that once a globally-ordered 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. 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.

Globally-ordered 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 [7].

We chose to use a three-phase protocol because it is non-blocking: replicas in the new majority view are able to continue processing client requests. Having a non-blocking protocol is important because replicas cannot perform queries while a globally-ordered update is running. A two-phase protocol would require fewer messages, but an inopportune failure would prevent the entire system from processing queries.

5. System Issues

Our method is generic and can be instantiated to provide a particular service. The instantiator provides what is essentially an abstract data type with a procedure for each update and query operation of the service. He or she need be concerned only with the implementation of the unreplicated object; all details of replication are taken care of automatically.

The instantiator needs to define the class of each operation, i.e., whether it is client-ordered, server-ordered, or globally-ordered. The easiest way to do this is by operation name, e.g., send_mail is client-ordered, while add_user is server_ordered. However, finer groupings are possible in which both the operation name and the input arguments are considered; such groupings can be defined by providing a procedure that would be called at runtime to determine the class of a particular call.

To create an application service, it is necessary to decide how many replicas there are and where they reside. A service can be adjusted to changes in load (e.g., as clients are added to the system, or as the system grows) by reconfiguration. This is accomplished by carrying out what is essentially a view change [8] during which replicas can be added and removed from the service. An adaptation of the view change algorithm to our multipart timestamp replication scheme is described in [15].

In the remainder of this section we discuss the two main issues that determine the effectiveness of our technique in large systems: the cost of replication, and the cost of managing the timestamps returned by replicated services.

5.1. Cost of Replication

In our scheme the service nodes are disjoint from the client nodes, which means that the number of replicas is independent of the number of clients. Since typically there will be large numbers of clients, this is an important consideration. Having fewer 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).

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. This latter cost is a problem only if executing an update is expensive. In this case, it may be possible to reduce the cost of processing an update received in gossip by doing "value logging". In this scheme, if an update request is ready when it arrives from the client (and most updates will be ready, just as most queries are ready when they arrive), the receiving replica would perform it and log a description of its effect on *val*. Other replicas would simply incorporate the logged effect into their *val* instead of performing the update. This technique will be worthwhile provided the amount of information to be logged is small and the cost of incorporating that information into *val* is less than performing the update.

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. Such a service still appears to be a single entity to clients, but better performance can be achieved by partitioning.

With partitioning, the replicas can be 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; the front end for such a system would probably maintain information about the partitioning so that a client request could be sent to one of the replicas in the request's group. The timestamps would contain components for all replicas; this is necessary so that the client can control the ordering of operations across partitions. However, the timestamp components for other partitions are 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. In addition, gossip is exchanged only among group members.

Partitioning can reduce the amount of gossip and update processing, but such a system may have large timestamps. The size of timestamps can be reduced by distinguishing between *read-replicas* and *write-replicas*. Read-replicas handle only query operations. Write-replicas perform updates and timestamps contain entries only for them. Each write-replica is responsible for sending new information to some subset of the read-replicas; to compensate for the failure of a write-replica, these subsets could overlap, or some reconfiguration scheme could switch the responsibility to another write-replica temporarily. In addition, a read-replica could request recent information from a write-replica if desired.

In essence this scheme places read-only caches at various convenient locations in the network, e.g., one in each local area net. It is suitable for a system in which updates are relatively rare but queries are frequent.

Another possibility is to use a hierarchical approach. This approach is valid only when all client interaction that exposes information about the service state happens through calls on service operations. 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 [17]; in this application, a client's query depends only on 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 [9].

5.2. System Structure

Up to now we have considered services in isolation; now we consider a system containing many services. In such a system there will be many different multipart timestamps, one kind for each service, and it will sometimes be necessary to distinguish them. This is easily done by having each timestamp identify its service.

The service interfaces described so far do not allow the operations of different services to be ordered relative to one another. An *open* service supports inter-service operation ordering; an *isolated* service does not provide this ability. The operations of an open service need extra parameters because we must be prepared to send and receive label timestamps for other services in these calls:

update (prev: label, others: set[label], op: op) **returns** (uid)
query (prev: label, op: op) **returns** (newl: label, new_others: set[label], value: value)

An update operation takes in labels for other services in addition to the label for its own service; a query operation returns labels for other services in addition to its own. By including a label L_T for some service T in a call to an operation *o* of service S, a client is indicating that *o* must be ordered after all the T operations identified by L_T.

When a query (perhaps by a different client) observes the effects of *o*, it will return a label for T that identifies all updates identified by L_T (and possibly some additional ones). This allows the client to require that a later call on a T operation be ordered after all updates in L_T. (An open system is similar to a partitioned one as discussed in Section 5.1 in the sense that the the labels for other services correspond to the parts of the timestamps for the other partitions.)

The implementation of an open service is also a little different from what was discussed earlier. Each replica maintains a list of labels for other services. When it processes an update, it merges its labels with the corresponding ones in the list. When a query returns, it returns the list of labels in addition to its result. Just as a query need not return *val* but instead can return a result computed using *val*, so it need not return the entire list, but instead just some of the labels in the list.

Services are often *encapsulated* in the sense that they are used by a number of predefined clients. In fact, the most common situation we have observed is an isolated service used by a single predefined kind of client. The timestamps of an encapsulated service are used only by its clients and are never visible to any other part of the system. For example, the part of a system that implements remote procedure calls might use a location service to determine the location of the called module. Clients of the RPC service know nothing about the location service, and never need to manipulate its timestamps.

Encapsulated services raise no general timestamp management problem; the vast majority of programmers need never be concerned with their timestamps. The same is not true for unencapsulated services, however. One possibility is to provide implicit management of their timestamps; each client front end maintains a list of timestamps (just like the replicas of an open service), sends all timestamps in all messages to other clients and in calls to updates of open services, and merges all received timestamps with the corresponding ones in its list. Implicit management would be costly if there were many unencapsulated services, however. An alternative is for the person implementing a client to deal with timestamps explicitly, i.e., the client program explicitly specifies what timestamps should be sent in what messages. Performance will be better with the explicit approach primarily because timestamps will need to be sent in fewer messages. Furthermore, we believe it is not difficult to decide what timestamps to send in what messages, because the needed information will be contained in the specifications of the modules with which the client interacts.

6. Related Work

Our work builds on numerous previous results in the area of highly available distributed systems and algorithms, including general replication techniques such as voting [11, 13, 2] and the primary copy method [1, 27, 26]. Our work is also related to gossip schemes [12, 10, 34]. In this section, we focus on the most closely related work, namely providing high availability for applications where operations need not be ordered identically at all replicas. In this light, we compare our method with the relevant gossip schemes and with the work on ISIS. We also consider approaches where consistency is relaxed in order to improve performance.

Some gossip methods [10, 34] require that a replica of the service exist at every client node, leading to increased storage requirements and message traffic. Furthermore, these methods are application specific and do not allow the operation order to be controlled by clients.

In the replication method used in the implementation of the Grapevine system [6], service nodes are distinct from client nodes, operations are performed at exactly one replica, and updates propagate in the background to other replicas. However, in Grapevine consistency is sometimes sacrificed to improve performance, leading to undesirable behavior such as the existence of distinct users with the same name.

The sweep-based replication method used in the design of Lampson's global-name service [19] was developed to address some of the shortcomings that existed in Grapevine [6] while satisfying the original goals of fast response time and high availability. As in Grapevine, the name-service client operations interact directly with one replica; the method for spreading updates to all replicas is to periodically perform a global "sweep" operation, enforcing the same total order on all updates that preceded the sweep. However, the order induced may be inconsistent with the order observed by clients, and furthermore, no consistency is guaranteed for operations that executed after the last sweep. In contrast, our method allows clients to prevent such inconsistencies completely and without degrading the performance of operations that require the weaker (partial) ordering.

ISIS [4, 3] provides a replication method that supports causal order but in a different fashion with its CBCAST broadcast method. Instead of using timestamps, ISIS piggybacks all CBCAST messages received by a node on all messages sent by a node and in this way ensures that needed updates are known at the replicas. The advantage of the ISIS technique is that queries never need to be delayed. However, our system is more efficient in terms of information that must be remembered and the size of messages. Our messages are much smaller since they contain timestamps rather than the messages these timestamps identify. Also, we avoid a major system wide garbage collection problem that exists in ISIS, namely, knowing when it is safe to discard information about old messages. Futhermore, unlike ISIS, our method works in wide area nets and in the presence of network partitions.

In addition to CBCAST, ISIS provides two other broadcast protocols, ABCAST and GBCAST (counterparts to our server-ordered and globally-ordered operations). Our implementation of these operations is more efficient than ISIS's implementation of ABCAST and GBCAST. The processing of a server-ordered operation requires only a majority of service replicas, providing improved availability and response time compared to ABCAST. Our implementation of globally-ordered operations is also more efficient. The synchronization part of the ISIS protocol that makes sure that the current GBCAST is executed in the same state everywhere requires that all nodes in the network participate in the synchronization part of the protocol. In contrast, our method uses fewer messages and requires that only the service nodes participate in the global synchronization.

The Psynch protocol [29] is an IPC mechanism that supports causal message ordering by explicitly encoding this ordering in each message. It operates in the presence of network and process crashes and can be viewed as an optimized implementation of ISIS CBCAST that transmits message ids instead of messages and thus cuts down the costs of the message traffic and message garbage collection. A recent paper [25] describes a way of using the protocol to implement replicated services; it supports two kinds of operations, commutative operations (where the order of execution does not matter), and totally ordered operations, so it is less general than our method.

The recent "bypass" implementation of ISIS [5] has adopted multipart timestamps to reduce the inefficiencies in their original method. In ISIS a system is composed of a number of "process groups". Two process groups can communicate only if they have a member in common, which means that client and server groups must overlap. For example, there might be a single group containing all clients and the service replicas. Alternatively, if clients do not communicate with one another, there could be a separate group for each client consisting of the client process and a process for each service replica. Each group has its own multipart timestamp, and when communication occurs the message is multicast to all members of the target group. Like our system, this new ISIS implementation trades occasional delay in message processing for smaller messages and easier garbage collection.

In the "straightforward" bypass implementation, every message would contain a timestamp for every group in the system. Even if we also send a timestamp for every service in every message, our implementation would perform better. Our timestamps are smaller since they only contain components for the service replicas; in ISIS, timestamps must contain components for clients as well. In addition, we send fewer messages than ISIS because it broadcasts each operation call to all group members; in our scheme replicas communicate using gossip messages that do not include information about query operations and can contain information about many different update calls. Furthermore, if ISIS uses the structure of having each client in a separate group with the service replicas, it will send many more timestamps in messages, since its messages must contain a timestamp for each such group; we need to include only the service timestamp.

To reduce the size and number of messages, the ISIS bypass paper describes some alternative mechanisms. For example, it discusses a method for performing analysis of a graph representing communication patterns in a system to determine when optimizations are possible, but the analysis method is limited; many systems would not be considered optimizable using their analysis. Therefore, the paper also discusses some protocols that can reduce number and size of messages by delaying certain activities (e.g., when a client can send the next message).

In addition to the differences in detail discussed above, there is also a difference in philosophy between our system and ISIS. As discussed in Section 5, we expect builders of applications (and services) to decide explicitly what to do with the uids and labels returned by calls to service operations. Clients should communicate labels only when it matters to what they are doing. In fact, a person will also have to decide what to do in ISIS in order to improve performance.

7. Conclusion

This paper has described a new, lazy replication method that allows application semantics to be taken into account to weaken implementation constraints. Three kinds of operations are supported: client-ordered operations, server-ordered operations, and globally-ordered operations. The method is generic and can be instantiated to provide a particular service. The instantiator need be concerned only with the implementation of the unreplicated object; all details of replication are taken care of by the our method.

For applications that can use it, the method provides good availability and response time with low communication costs. The method is optimal in the sense that it does not introduce unnecessary synchronization among client-ordered operations and therefore does not delay the processing of queries. The method is particularly suitable for applications in which most of the update operations are client-ordered. The other two orders are important because they increase the applicability of our approach: They allow it to be used for applications in which some operation calls require a stronger order than the client-specified order. However, they must be used sparingly. If most calls are server-ordered, our method will perform similarly to techniques such as voting [11] and primary copy [28]. If many calls are globally-ordered, our method will not perform well. Globally-ordered calls are expensive because this is where we pay for the good performance of the client-ordered calls.

When confronted with the need for a highly available service, a designer has a limited number of choices. One possibility is to trade consistency for performance as in Grapevine [6]. Another is to use standard atomic methods [27, 11, 2, 13]; with these methods operations really happen by the time they return, so there is no need to carry information (e.g., timestamps) in messages to indicate ordering constraints. Like the atomic methods, our approach provides consistency, but ordering information is needed because updates happen lazily. Our method is a good choice provided the size of the extra information in the messages is small (i.e., there is a modest number of write-replicas) and provided most operations can take advantage of the laziness (i.e., client-defined ordering is appropriate).

8. Acknowledgments

We wish to thank Boaz Ben-Zvi, Phil Bernstein, Andrew Black, Dorothy Curtis, Joel Emer, Bob Gruber, Maurice Herlihy, Paul Johnson, Elliot Kolodner, Murry Mazer, and Bill Weihl 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.
In *Proc. of the 2nd International Conference on Software Engineering*, pages 627-644. October, 1976.
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):596-615, December, 1984.

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

[4] Birman, K., and Joseph, T.
Exploiting Virtual Synchrony in Distributed Systems.
In *Proc. of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123-138.. November, 1987.

[5] Birman, K., Schiper, A., Stephenson, P.
Fast Causal Multicast .
Technical Report TR 90-1105, Cornell University, Dept. of Computer Science, Ithaca, N. Y., March, 1990.

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

[7] El-Abbadi, A., and Toueg, S.
Maintaining Availability in Partitioned Replicated Databases.
In *Proc. of the Fifth Symposium on Principles of Database Systems*, pages 240-251. ACM, 1986.

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

[9] Farrell, K. A.
A Deadlock Detection Scheme for Argus.
1988.
Senior Thesis. Laboratory of Computer Science, M.I.T., Cambridge, MA.

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

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

[12] 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.

[13] Herlihy, M.
A Quorum-consensus Replication Method for Abstract Data Types.
ACM Trans. on Computer Systems 4(1):32-53, February, 1986.

[14] Herlihy, M. P., and Wing, J. M.
Axioms for concurrent objects.
In *Proc. of 14th ACM Symposium on Principles of Programming Languages (POPL)*, pages 13-26. January, 1987.
Also CMU-CS-86-154.

[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 3:393-420, 1988.

[17] Ladin, R.,
A Method for Constructing Highly Available Services and An Algorithm for Distributed Garbage Collection.
PhD thesis, M.I.T. Laboratory for Computer Science, May, 1989.

[18] Lamport, L.
Time, Clocks, and the Ordering of Events in a Distributed System.
Comm. of the ACM 21(7):558-565, July, 1978.

[19] Lampson, B. W.
Designing a Global Name Service.
In *Proc. of the 5th Symposium on Principles of Distributed Computing*, pages 1-10. ACM SIGACT-SIGOPS, August, 1986.

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

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

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

[23] Liskov, B.
Distributed Programming in Argus.
Comm. of the ACM 31(3):300-312, March, 1988.

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

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

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

[27] 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.

[28] Oki, B.
Reliable Object Storage to Support Atomic Actions.
Technical Report MIT/LSC/TR-308, M.I.T. Laboratory for Computer Science, Cambridge, MA, 1983.

[29] Peterson, L. L., Bucholz N. C., and Schlichting, R.D. .
Preserving and using context information in interprocess communication .
ACM Trans. on Computer Systems 7(3):217-246, August, 1989.

[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-9:240-247, May, 1983.

[31] Schlichting, R. D., and Schneider, F. B.
Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems.
ACM Trans. on Computing Systems 1(3):222-238, 1983.

[32] 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.

[33] Weihl, W.
Distributed Version Management for Read-only Actions.
IEEE Trans. on Software Engineering, Special Issue on Distributed Systems 13(1):55-64, 1987.

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

Table of Contents

List of Figures