Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems

THOMAS A. JOSEPH and KENNETH P. BIRMAN
Cornell University

Many distributed systems replicate data for fault tolerance or availability. In such systems, a logical update on a data item results in a physical update on a number of copies. The synchronization and communication required to keep the copies of replicated data consistent introduce a delay when operations are performed. In this paper, we describe a technique that relaxes the usual degree of synchronization, permitting replicated data items to be updated concurrently with other operations, while at the same time ensuring that correctness is not violated. The additional concurrency thus obtained results in better response time when performing operations on replicated data. We also discuss how this technique performs in conjunction with a roll-back and a roll-forward failure recovery mechanism.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *distributed databases*; C.4 [**Computer Systems Organization**]: Performance of Systems—*reliability, availability, and serviceability*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *synchronization*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; *fault-tolerance*: H.2.2 [**Database Management**]: Physical Design—*recovery and restart*; H.2.4 [**Database Management**]: Systems—*transaction processing*.

General Terms: Reliability

Additional Key Words and Phrases: Replicated data, concurrent update, piggybacked update, roll-forward recovery.

1. INTRODUCTION

The advent of distributed computing systems has added a new aspect to fault tolerance. A distributed system consists of a number of processing units (sites) connected by a communications network and fault tolerance can hence be achieved by replicating information at multiple sites. This, however, usually entails a substantial performance degradation, because the sites must coordinate and synchronize their actions to ensure the consistency and correctness of replicated information. The associated overhead manifests itself as poorer response time. In certain situations, an application may not be able to tolerate this delay, and in any case, it is usually desirable to improve responsiveness to the

This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA-903-85-C-0124. The views, opinions, and findings contained herein are those of the authors and should not be construed as an official DoD position, policy, or decision. This work was also supported in part by a grant from the Sperry Corporation.
Authors' address: Department of Computer Science, Cornell University, Ithaca, NY 14853.

extent possible. In this paper, we present a technique that reduces, and sometimes eliminates, the latency arising from replication. It works by relaxing the level of synchronization usually employed to maintain the consistency of replicated data, while at the same time ensuring that this consistency is not compromised. In a system where the cost of managing replicated data is dominant, the additional concurrency thus obtained will result in better response time.

The technique we describe is applicable to distributed computer systems that are asynchronous in nature, that is, they coordinate their actions by sending messages to one another and do not use a "global clock" for synchronization. We assume that the communication medium is reliable, and that processors are *fail-stop* [15]: the only failure they suffer is a halting failure—they stop processing, and the other sites are informed of their failure[1]. No information stored at a site survives a failure. Although we describe the method with respect to replicated databases, it is applicable to more general replicated systems. The generalization to object-based distributed systems is described in [11].

In the next section, we describe our model for replicated systems. It is similar to the model for replicated databases in [2]. In Section 3, we examine the concept of "order" as applied to an asynchronous system, and draw some conclusions about the message patterns in such systems. We next describe how replication introduces delays in performing operations. In Section 5, we present an algorithm that could be used to eliminate some of the latency caused by replication, but which is unsatisfactory because it generates unnecessarily heavy message traffic. The algorithm of Section 5 is then modified to form the basis of a method to increase concurrency and thus improve response time. Sections 7 and 8 discuss how this scheme performs in the presence of site failures. An implementation of this method and its performance are described in the next section, and Sections 10 and 11 conclude the paper.

2. MODEL

A replicated system is modeled by a collection of *replicated data items* that are accessed by *logical operations*. A replicated data item is one that is stored redundantly at multiple sites[2]. The operations that can be performed on a replicated data item are *read*, which returns its value, and *write*, which changes its value.[3] Although each *copy* of a data item can be read or written independently, a user has the view that only a single copy exists. A logical operation issued at a site is implemented by executing a *physical operation* on one or more copies of the data item in question. For now, we assume that a copy always exists at a site where a logical read is issued, and that it is implemented by a physical read of that copy. Hence, we do not distinguish between a logical read and the physical read that implements it—both terms refer to the same operation. A logical write,

[1] This abstraction can be implemented on top of an unreliable communication medium by an appropriate software layer.

[2] The work in this paper is equally valid if the term *site* is interpreted as *process at a site*, provided processes do not share memory, and communicate only by sending messages to one another.

[3] Write operations are not limited to operations that overwrite the value of a data item with a new one. They may include such operations as incrementing a data item, inserting an element in a data item that represents a queue, etc.

on the other hand, results in an *update instruction* being sent to each of the copies, and physical writes occur on all of them.

High level operations are modeled by *transactions*, which perform operations on different data items in a certain order. A transaction T is modeled by a tuple $(OP_T, <_T)$, where OP_T is a set of logical operations and $<_T$ is a partial order on them. The partial order $<_T$ embodies the data flow relationships between the operations in OP_T, and a correct execution of T must observe this order. A logical read (write) operation on data item x by transaction T is represented as $R(x, T, s)$ (resp. $W(x, T, s)$), where s is the site at which the operation is issued. We use the notation $O(x, T, s)$ to represent an arbitrary read or write operation, and use R, W, or O when the context is clear. If a transaction has more than one read (write) on the same data item, the ith read (write) is denoted as R^i (resp. W^i).

When transactions are run concurrently, the execution of operations from different transactions may be interleaved. A correct execution should respect the partial order $<_{T_i}$ for each transaction T_i. In addition, the execution should be *serializable*, that is, its effect should be the same as if the transactions were executed independently in some serial order. A *concurrency control algorithm* is used to ensure that all executions are serializable. A number of such algorithms are known and are discussed in [1, 3, 12, 14]. Here we assume that a conflict-based algorithm is in use, although the results can be extended to other kinds of algorithms. Two logical operations *conflict* if they both operate on the same data item and at least one of them is a write. A conflict-based algorithm works by ordering conflicting operations in such a way that only serializable executions are produced. The mechanisms used to order operations depend on the degree of synchronization available in the system. In the next section, we discuss the concept of "order" as it relates to an asynchronous system. The details of the particular algorithm used are not essential to our treatment.

Formally, the concurrent execution of a set of transactions $T_1, T_2, \cdots, T_n$ gives rise to a *logical log* $L = (OP_L, <_L)$. OP_L, is the union of OP_{T_i} for all i, and $<_L$ is a partial order on the operations in OP_L, representing the order in which they are executed. If $O_1 <_L O_2$, we say that O_2 is *scheduled after* O_1. L is *correct* if $\forall i$: $<_L \supseteq <_{T_i}$, and if there exists a total order $<_S$, called a *serialization order*, on the set $\{T_1, T_2, \cdots, T_n\}$ such that if an operation O_1 in transaction T_a conflicts with an operation O_2 in T_b, and if $T_a <_S T_b$, then $O_1 <_L O_2$.

Given a correct logical log L, we say that a write operation W *writes before* an operation O on the same data item, if O is scheduled after W and there is no other write operation W' on the same data item scheduled after W but before O. If O is a read operation, it means that O reads the value written by W. In other words, $W(x, T_a, s)$ writes before $O(x, T_b, t)$ if $W(x, T_a, s)$ and $O(x, T_b, t)$ belong to OP_L, $W(x, T_a, s) <_L O(x, T_b, t)$, and there is no $W(x, T_c, v)$ such that $W(x, T_a, s) <_L W(x, T_c, v) <_L O(x, T_b, t)$.

In Figure 1, we show three transactions, T_a, T_b, and T_c, and a correct logical log L for these transactions. There are three sites s, t, and v in the system, and the data items x, y, and z each reside at all the sites. The partial order $<_L$ is the transitive closure of the relations shown in the figure. A serialization order S corresponding to L is $T_b <_L T_c <_L T_a$. $W(x, T_c, v)$ writes before $R(x, T_a, s)$. $W(y, T_c, v)$ writes before $R(y, T_a, s)$, and $W(x, T_b, t)$ writes before $W(x, T_c, v)$.

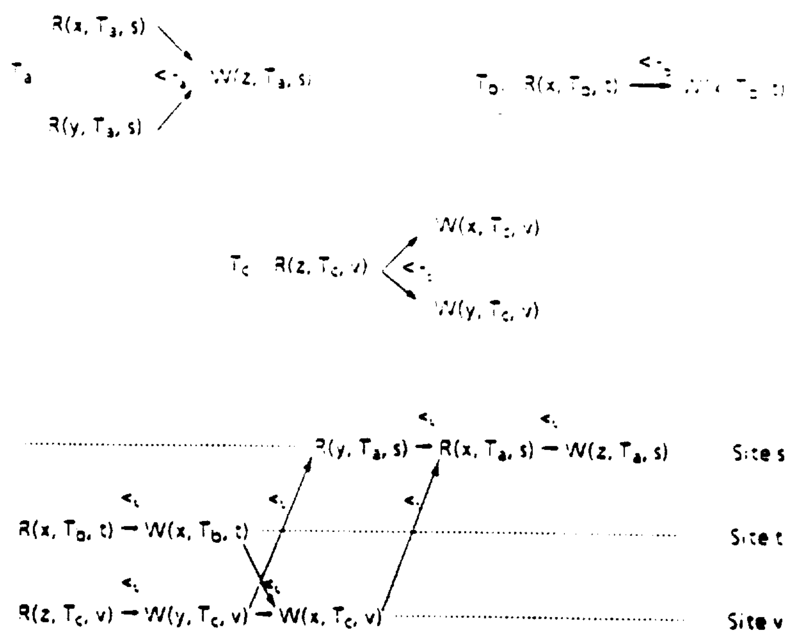

Fig. 1. Three transactions and a correct logical log.

3. ORDER IN AN ASYNCHRONOUS SYSTEM

A logical log depicts a temporal order between executions of logical operations. As Lamport has observed [13], the only temporal order that is meaningful between events at different sites in a distributed asynchronous system is one based on the messages sent between them. Events that occur at a single site are ordered in time in the normal way. The act of sending a message from one site precedes the receipt of the same message at another. Any two events that are not (transitively) related to each other in one of these two ways are said to be concurrent, that is, there is no temporal order between them. Informally, an event a at a site s is ordered temporally after an event b at site t if, and only if, there is a sequence of messages, the first one originating from site t after the event b, the next message being sent from the destination site of the first message after the first message is received there, and so on, with the last message being received at site s before the event a. If there is no such "path" of messages from b to a, b can not causally affect a, and the two events are not temporally ordered. In particular, for a concurrency control algorithm to enforce an order between logical operations at different sites, it must cause *synchronization messages* to be sent between the sites. This is reflected in the *physical log*.

A physical log P_L for a logical log L is a record of all logical and physical operations carried out at each site, together with message transmissions between sites. It consists of a set of *actions*, where an action is the execution of a logical or a physical operation, or the sending or receipt of a message. Additionally, it records the order in which actions are carried out. We use upper case letters to represent logical operations and lower case for physical operations. Thus, a physical write operation at site t resulting from a logical operation $W(x, T_a, s)$ is denoted $w(x, T_a, t)$. $R(x, T_a, s)$ and $r(x, T_a, s)$ refer to the same action—the read operation on x. We use the notation send(m) (receive(m)) for the action corresponding to sending (receiving) message m, and the notation sender(m) (receiver(m)) to denote the site from (at) which m is sent (received).

Every physical log P_L induces a temporal order $\rightarrow_L$ on its actions. As in [13], this is defined as the transitive closure of the following relations:

(1) If a and b are two actions at the same site, and a occurs before b, $a \rightarrow_L b$.
(2) If a is send(m), and b is receive(m), $a \rightarrow_L b$.

A physical log P_L represents the implementation of the behavior described in a logical log L. Thus for a physical execution to be correct, the physical log must contain all the orderings that are in the logical log. Since $\rightarrow_L$ contains all possible temporal orderings between actions in P_L, it follows that $\rightarrow_L \supseteq <_L$. Also note that $\rightarrow_L$ is consistent with (global) real time, that is, if $a \rightarrow_L b$, then a occurs before b in real time (but not necessarily *vice versa*). This implies that $\rightarrow_L$ is acyclic, otherwise it would mean that an event could occur before itself in real time.

Observation. A *message path* is said to exist from an action a at site s to an action b at site t if the physical log contains a sequence of messages $m_1, m_2, \cdots, m_n$ $(n \geq 0)$ such that sender(m_1) $= s$, $a \rightarrow_L$send(m_1), receiver(m_i) = sender(m_{i+1}), receive(m_i) $\rightarrow_L$ send(m_{i+1}), receiver(m_n) $= t$, and receive(m_n) $\rightarrow_L b$. (If $s = t$, then $n = 0$ and the path is empty.) Observe that it follows from the definition of $\rightarrow_L$ that $a \rightarrow_L b$ if, and only if, there is a message path from a to b.

THEOREM. *For any correct logical log L that contains two operations $W(x, T_a, s)$ and $O(x, T_b, t)$, such that $W(x, T_a, s)$ writes before $O(x, T_b, t)$, the physical log P_L contains a message path from $W(x, T_a, s)$ to $O(x, T_b, t)$.*

PROOF. Since $W(x, T_a, s)$ writes before $O(x, T_b, t)$, $W(x, T_a, s) <_L O(x, T_b, t)$. Because $\rightarrow_L \supseteq <_L$, $W(x, T_a, s) \rightarrow_L O(x, T_b, t)$. The result follows from the observation above.

4. DELAY INTRODUCED BY REPLICATION

If an operation W writes before an operation O in a nonreplicated system, the value of the data item at the time O is executed is always the value written by W. In a replicated system, if $W(x, T_a, s)$ writes before $O(x, T_b, t)$, the system must provide the same effect as if x were nonreplicated [1]. In the simplest implementation of logical writes, the value to be written is broadcast to all sites where a copy of the data item resides. A physical write occurs at those sites, and then a confirmation message is returned to the site where the logical write was issued. Only then is the logical write considered completed. This solution is unsatisfactory because every write operation entails waiting for responses before the operation after the write can proceed. Figure 2 depicts this latency.

We present an implementation of logical writes that permits an operation after a write to proceed as soon as possible, with the physical writes being performed concurrently at the other sites. We do this in two stages. First, we describe an

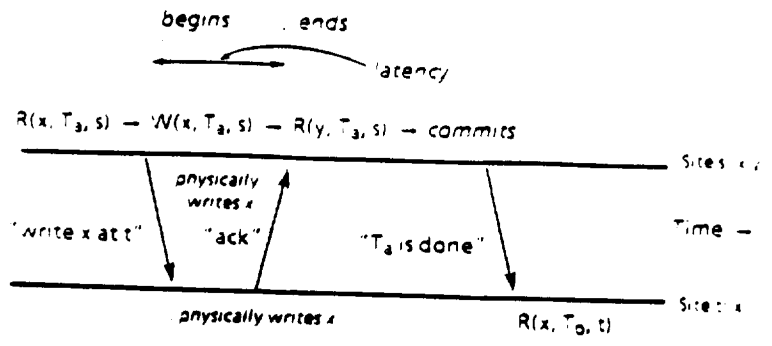

Fig. 2. Latency resulting from distributed write.

algorithm that piggybacks update instructions on messages used by the concurrency control algorithm for synchronization. Then, the algorithm is modified to permit updates to be broadcast concurrently. This gives the desired result.

5. PIGGYBACKED UPDATE

Piggybacked update is an implementation of logical writes that relaxes the level of synchronization by deferring physical write operations until their results are actually needed. A logical write is considered completed when the required update instructions are generated (but not necessarily sent). This eliminates the latency described in the last section. An added advantage is that update instructions are all piggybacked on messages required for concurrency control. This reduces the message traffic in the system.

If $W(x, T_a, s)$ writes before $O(x, T_b, t)$, the physical writes for W can be deferred, provided that w is carried out before o everywhere, and that no other physical write w' occurs between w and o. Consider the update instruction that must travel from site s to site t, where a physical write w must occur. The theorem in Section 3 shows that there is always a sequence of synchronization messages upon which this update instruction can be piggybacked such that it will arrive at t before O is performed. The implication is that it is never necessary to perform a broadcast to distribute updates; they can all be piggybacked upon synchronization messages. This motivates the piggybacked update algorithm shown in Figure 3 and described below.

The piggybacked update algorithm assumes that a concurrency control algorithm is in operation, which imposes an order on the execution of logical operations by sending synchronization messages between sites.[4] Copies of the update instructions resulting from a logical write at a site s are piggybacked on *all* synchronization messages that are subsequently sent from s. By the theorem, the update instructions will arrive at their destinations before the execution of any operation that depends on it. The algorithm requires that each site s maintain a buffer $Outgoing_s$ of outgoing update instructions. $Outgoing_s$ contains update

[4] It is convenient to think of operations is at the same site as being ordered by synchronization messages sent from the site to itself, though this need not be actually implemented.

Whenever a synchronization message m is being sent from site s to site t:

— Piggyback on m a copy of all update instructions in $Outgoing_s$, in order.
— Piggyback the values of $LastSeen_s$ and $TheirView_s$.

When a synchronization message is received from site t:

— For each site v, accept all piggybacked update instructions originating from v whose timestamps are greater than $LastSeen_s[v]$, and set the value of $LastSeen_s[v]$ to the largest such timestamp.
— Process, in order, all updates pertaining to copies of data items at s, and append all other update instructions to $Outgoing_s$, preserving their order.
— Set the values of $TheirView_s[t][v]$ to the piggybacked values of $LastSeen_t[v]$.
— Set the value of $TheirView_s[v][w]$ to the larger of $TheirView_s[v][w]$ and the piggybacked value of $TheirView_t[v][w]$.
— Delete from $Outgoing_s$ all updates from site w to site v with timestamps smaller than or equal to $TheirView_s[w][v]$.

Figure 3: Piggybacked update algorithm, as followed by site s.

instructions from site s to other sites, as well as updates[5] that pass through site s en route to other sites. Updates in $Outgoing_s$ are ordered according to the time they are first received by site s. Copies of update instructions remain in $Outgoing_s$, and continue to be piggybacked on outgoing synchronization messages, until site s learns that the destination has already received a copy. If multiple copies of an update arrive at a site, all copies except the first are ignored.

To detect whether a copy of an update has reached its destination and to identify and ignore duplicate copies of the same update, each logical write operation is given a *timestamp*. Timestamps have the property that if W and W' are two logical writes at the same site, and if $W <_L W'$, then the timestamp of W' is strictly greater than that of W. The timestamp of a logical write, with the site name appended, is called its *operation-ID*. Note that timestamps and operation-ID's can be generated locally at each site. Each update instruction carries the timestamp of the logical write it corresponds to. It follows from the way in which updates are piggybacked that if an update from a site v arrives at a site s by some path of synchronization messages, then all updates from v to s with smaller timestamps must have already arrived at s. They, too, would have been piggybacked on the same message path, unless they had already reached s by another path.

At each site s, the array element $LastSeen_s[v]$ records the value of the largest timestamp that site s has observed on updates resulting from a logical write at v. It ignores any update originating from v that carries a smaller timestamp, because it must have already received a copy. Additionally, $TheirView_s[v][w]$ records the value of $LastSeen_v[w]$ at the time of the last message that site v sent to site s. Site s deletes from $Outgoing_s$ any update from w to v carrying a smaller timestamp than $TheirView_s[v][w]$, because v must have already received a copy of it.

We now show that the algorithm is correct. We need to show that if $W(x, T_a, s)$ writes before $O(x, T_b, t)$, then all physical writes w occur before the physical operation o at that site. (If O is a read, this happens only at site t.) We must also show that no other physical write w' corresponding to, say $W'(x, T_c, v)$ occurs between w and o.

[5] We use the term *update* to mean *update instruction*.

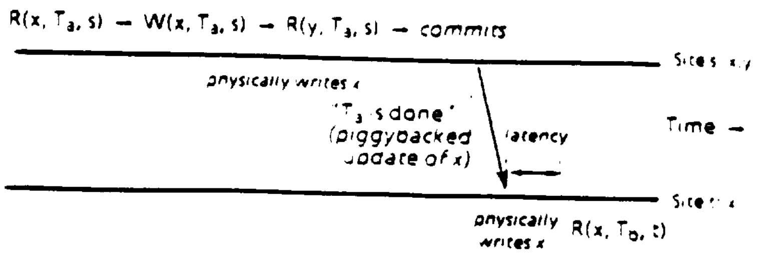
R(x, T3, s) → W(x, T3, s) → R(y, T3, s) → commits
Sites x, y
physically writes x
"T3 is done" (piggybacked update of x)
latency
Time →
Site t: x
physically writes x
R(x, Tb, t)

Fig. 4. Reduction in latency using piggybacked update.

Because there is always a message path from W to O (Theorem, Section 3), the algorithm guarantees that a copy of the update instruction for W will arrive at site t before O is executed. If O is a read, this is sufficient to ensure that w occurs before o. If O is a write, the update instruction for W will be piggybacked on any message that carries the update for O,[6] and will be ordered before the update for O. This means that w will be performed before o at any site where o is performed.

To show that w' will not occur between w and o, note that W and W' conflict, so it follows from the definition of serializability that they must be ordered relative to each other. Thus either $W' <_L W$ or $W <_L W'$, depending on the serialization order. In the first case, there is a message path from W' to W, and a copy of the update instruction for W' will arrive at t before W is executed. Hence, the update instruction for W' will be piggybacked on any message carrying the update for W, and ordered before it. As a result, w' will be performed before w, and not between w and o.

The second case ($W <_L W'$) is proved as follows. O and W' conflict, so $O <_L W'$ or $W' <_L O$. Since W writes before O and $W <_L W'$, the only possibility is that $O <_L W'$. If O is a write operation, the same argument as above shows that any physical write o occurs before w'. If O is a read operation, note that $O <_L W'$ implies that there is a message path from O to W'. There is a message path from W' to w', along which the update instruction travels. Now if w' were to occur before the physical read o, there would be a cycle in the physical log, which is impossible. (In this case O and o represent the same event.) Thus w' does not occur between w and o.

We have shown that the piggybacked update algorithm results in an implementation in which physical operations respect all the dependencies between logical operations. It does not require that physical writes on different copies of the same item be synchronized, and thus reduces the latency associated with executing a logical write operation. It does not depend on the actual details of the concurrency control algorithm, provided that some mechanism is used to order conflicting operations. In Figure 4, we show how the latency depicted in Figure 2 is reduced by piggybacked update.

The algorithm has the obvious disadvantage that update instructions are piggybacked on message paths that might never lead to the required destination, hence requiring unnecessarily large messages and buffers. Some of this overhead

[6] This is true unless a copy of the update instruction for W has already been received at its destination, in which case w will have already been performed before o.

could be eliminated if the concurrency control algorithm indicates which data items a particular synchronization message refers to. For example, with a lock-based concurrency control algorithm, the data items corresponding to a lock acquisition or release message are always known. In this case, update instructions for a data item could be piggybacked only on those synchronization messages that refer to it. Another obvious optimization is to not piggyback on a message to site t those updates in $Outgoing_s$ that have already been piggybacked on an earlier message to t. The buffer size can also be controlled by periodically broadcasting $LastSeen_s$ to other sites, enabling them to discard copies of updates that have already reached their destinations.

In many distributed systems the *number* of messages sent, and not their size, is an overriding cost factor. This is the case, for example, if messages are processed by a large number of software layers. In such a system, piggybacking would improve performance. However, as seen in Figure 4, piggybacked update could cause a large number of updates to be delivered at a site when a synchronization message arrives there. As a result, an operation that might have been waiting for a synchronization message to arrive suffers an additional delay while all the updates are processed. Note that this is a different kind of latency from the latency described earlier, as it does not involve waiting for messages to be sent and acknowledged, but is instead a wait for local processing to take place. The time for such local operations is usually much lower in magnitude than that for message transmissions. Moreover, this latency is not incurred unless there is an operation that is actually blocked, waiting for a write to be performed. However, even this latency can be avoided using *concurrent update*, which employs the piggybacked update algorithm as its basis.

6. CONCURRENT UPDATE

In the concurrent update scheme the piggybacking of operation-ID's is decoupled from the transmission of actual data. Operation-ID's are piggybacked on synchronization messages exactly as above, but update instructions are transmitted directly to destination sites, concurrently with ongoing operations. Such transmissions are done using *atomic broadcasts*, and are subject to a *broadcast ordering rule*.

An atomic broadcast has the following properties:

(1) The data broadcast are either received at all the destination sites or at none of them, even in the presence of site failures. Moreover, if broadcast B_1 is sent before broadcast B_2 by the same site s, then if B_2 is received (anywhere), B_1 is received by its destinations as well.

(2) If two atomic broadcasts made from the same site have destinations in common, the data are received at overlapping destinations in the same order that the broadcasts were initiated.

(3) If the data from an atomic broadcast B_1 are received at a site s before an atomic broadcast B_2 is initiated at s, then the data from B_1 are received before the data from B_2 at any overlapping destinations.

A number of protocols have been proposed for implementing broadcasts with these and similar properties [8, 16]. In [5], we describe a communiation subsystem in which this atomic broadcast is available as a primitive operation. Atomic broadcasts are included in a physical log in the same way as send events. We use the notation *AtBcast*(W) to represent the event corresponding to the initiation of an atomic broadcast for the update instructions of logical write W.

In the concurrent update scheme, a logical write $W(x, T_a, s)$ is considered completed once the physical write $w(x, T_a, s)$ is performed. The atomic broadcast to update the other copies of x may be initiated after an arbitrary amount of time, but must follow a broadcast ordering rule, which requires that if two conflicting logical writes are performed at the same site, then their atomic broadcasts are initiated in the same order that the logical writes were performed. In other words, if $W_1 = W(x, T_a, s)$ and $W_2 = W(x, T_b, s)$, and if $W_1 <_L W_2$, then $AtBcast(W_1) \rightarrow_L AtBcast(W_2)$. Physical writes are performed in the order in which atomic broadcasts are received at a site.

The execution of logical operations is governed by a *blocking rule*. The piggybacking of operation-ID's ensures that if a write operation $W(x, T_a, s)$ is scheduled before an operation $O(x, T_b, t)$, then W's operation-ID is received at t before O is performed. However, the update instruction for W may not have arrived, because *AtBcast*(W) can be deferred arbitrarily. The blocking rule requires that, in such a situation, the execution of O be blocked until the update instruction arrives, and the update is performed.[7]

The blocking rule ensures that if $W(x, T_a, s)$ writes before $O(x, T_b, t)$, then the physical write $w(x, T_a, t)$ is performed before O. If O is a read, this means that w occurs before o. If O is a write, property (3) of atomic broadcasts ensures that the update instructions for W will be received everywhere before those for O. Hence, w will be performed before o at all sites at which a copy of x resides. As with the piggybacked update scheme, we also need to show that no other physical write w' (corresponding to logical write W') will be performed between w and o. As before, we have two cases: $W' <_L W$ or $W <_L O <_L W'$.

If $W' <_L W$ and W' and W occur at the same site s, then the broadcast ordering rule requires that the atomic broadcast for W' be initiated before the broadcast for W. It follows from property (2) of atomic broadcasts that the update instructions for W' will be received everywhere before those for W. Hence, w' will occur before w and not between w and o. If W' and W occur at different sites, the piggybacked operation-ID for W' will be received at s before W is performed there. The blocking rule would cause W to wait until the update instruction for W' is received at s. Only then will the atomic broadcast for W be initiated. Property (3) of atomic broadcasts implies that the update instructions for W' will be received before those for W at any site where a copy of x exists. Hence, w' will be performed before w.

The second case arises if $W <_L O <_L W'$. If O is a write operation, and since $O <_L W'$, a similar argument to the one above can be used to show that o occurs before w'. If O is a read operation, the same argument as for piggybacked update shows that if w' were to occur before o, there would have to be a cycle in the

[7] An optimization is possible if O is a write operation that overwrites the value of x with a new value. In this case, O need not be blocked, and the update instruction for W is ignored when it arrives. This can be done by maintaining a list of operation-ID's of updates that must be ignored.

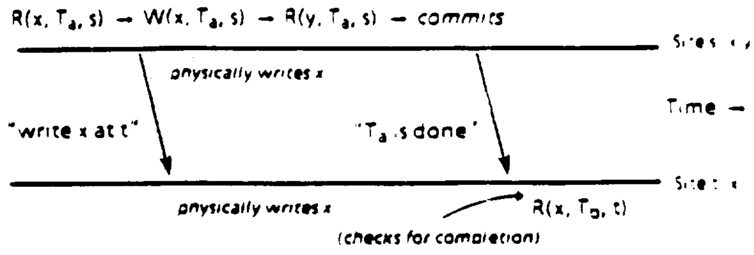

Fig. 5. Concurrent update.

physical log. Thus in this case too, w' cannot occur between w and o. This demonstrates the correctness of the concurrent update method.

We have shown how to implement logical writes in a way that permits physical updates to be carried out concurrently with other operations, instead of having to wait for updates to be acknowledged. The latency described in Section 4 does not occur. In addition, concurrent update does not piggyback (possibly large) updates on every synchronization message: only operation-ID's are piggybacked and operation-ID's are small. Hence, the overhead resulting from this method is low. Another advantage of transmitting updates concurrently is that physical writes are spread out over a larger period of time than with piggybacked update, where update instructions tend to arrive in bunches on synchronization messages. This could lead to better utilization of local processing capability. Figure 5 illustrates the effect of using this approach on the computation of Figures 2 and 4. If, as in this figure, sufficient time elapses before the results of a write are needed, no latency is incurred at all.

7. COMMITS, ABORTS AND FAILURES

In most database systems, a transaction terminates by executing a *commit* or an *abort* operation. Any physical writes performed before a commit or an abort do not cause a permanent change to the database; changes are effectively made to a copy of the variables. A commit causes the changes to be made visible to other transactions, while an abort leaves the database in a state that would have resulted if the transaction had not executed at all. Commit and abort operations can be included in our model by treating them as write operations on permanent copies of the variables accessed by a transaction. A commit writes a new value, while an abort writes the old one. A commit or an abort by a transaction conflicts with a read of any of the variables it writes, and is serialized by the concurrency control algorithm in the same way as a write operation. The previous results about concurrent update hence remain valid when commits and aborts are included.

One reason for replicating data is to ensure availability in the presence of site failures. In our model and in many others, all actions carried out at a site (e.g., physical writes, acquisition of locks) are lost if that site subsequently fails. When a failed site recovers, it copies the necessary data from an operational site. Since a failure results in a loss of information, many concurrency control algorithms (e.g., available copies [2]) handle failures by aborting any uncommitted transaction that has executed a logical operation at a site that fails. The transaction may then be restarted at another site. This form of recovery is often called *roll-back recovery*.

It may appear that concurrent update could be incorrect in the presence of failures because a piggybacked operation-ID could arrive at a site *s* en route to site *t*, and be lost there if site *s* fails. However, if an operation is scheduled for execution after a failure, there must be *some* path of synchronization messages leading to it from any operation that it depended on, otherwise, the execution would not be correct. The fact that operation-ID's are piggybacked on *all* synchronization messages ensures that they will arrive at their destinations by this path, even if operation-ID's along other message paths are lost.

Sometimes a commit is accompanied by an irreversible side effect, like dispensing money from a machine. If a commit does not have such a side effect, commit and abort instructions can be transmitted using concurrent update, in the same way as update instructions. Then if a transaction is committed at a site, and that site fails before performing the atomic broadcast of the commit information, all local information pertaining to the transaction is lost and the transaction is aborted at the other sites. The effect is as if the failure occurred during the commit operation, leading to an abort. On the other hand, if a local commit could have side effects that are not undone when a failure occurs, then all sites must engage in a commit protocol [10], which ensures that if a local commit occurs, then all sites commit. However, it is necessary to preserve the order of the commit operation relative to concurrent updates. For this reason, a *flush* must always be performed, that is, all outstanding updates must be broadcast atomically, before a commit protocol is invoked. With this change, concurrent update can be used in conjunction with a roll-back recovery method, such as the one previously described. Note that a flush is necessary only for operations that have irreversible side effects.

8. ROLL-FORWARD RECOVERY

Aborting transactions is just one way to recover from a site failure. Here we outline a *roll-forward* scheme, detailed in [6], and discuss how concurrent update can be used with it. In this scheme, a transaction is executed at more than one site, one of which, the *coordinator*, supervises its execution. The coordinator periodically takes checkpoints at the other sites, called its *cohorts*, by transmitting its local state to them. Each cohort carries a copy of the operations that must be performed as part of a transaction. Hence, a cohort can continue executing a transaction from a checkpoint, should the coordinator fail.

When a site fails, the operational sites execute a failure protocol, similar to the *exclude* transaction described in [2], during which they agree on the occurrence of the failure. They also agree on a relative time for the failure, effectively serializing the failure decision with respect to other operations. Hence, a failure decision appears in a physical log just as an operation on a data item. It is scheduled after operations that have been completed before the failure, and all other operations are scheduled after it. When the failure protocol ends, new coordinators are chosen for the transactions coordinated by the failed site, and they resume execution from their last checkpoint. The recovery scheme ensures that operations executed before a failure are not reexecuted, or if they are, they have exactly the same effects as before. This scheme can tolerate as many failures as there are cohorts. If more failures occur, we say that *total failure* has occurred, and in this situation, a transaction must wait until a site has recovered [17].

Checkpoints are included in our model by treating the checkpointed state of a transaction as a replicated data item, and the action of establishing a checkpoint as a write operation on this data item. A checkpoint CKP has a data flow dependency on all operations prior to it, and all operations subsequent to it depend on it. Thus, $O <_T CKP$ for all prior operations O in transaction T, and $CKP <_T O$ for all subsequent operations. For the purpose of the broadcast ordering rule, a checkpoint is considered to conflict with all write operations in T.

For this recovery scheme to be correct, it is necessary that the serialization order followed after a failure is the same as the one followed previously, because the effects of partially executed transactions may not be valid if a different order is followed. This requires that information pertaining to serialization decisions (acquisitions of locks, etc.) be maintained redundantly. Since writes occur at all copies of data items, replicating serialization information for write operations (e.g. acquiring write locks on all copies) is acceptable, but replicating information pertaining to read operations could make this scheme expensive.

How does concurrent update affect this recovery mechanism? Two aspects of the recovery method make it expensive: the need to transmit checkpoint information to cohorts and the need to replicate serialization information. When concurrent update is used, checkpoints and serialization information can be treated as replicated data items and can be updated concurrently. This makes a seemingly inefficient recovery mechanism viable.

It remains to be shown that concurrent update does not alter the relative order of dependent operations when a failure occurs. A logical operation carried out before a failure is said to be *known* after the failure if it occurs at a site other than the one that failed or if its atomic broadcast was completed before the failure. Operations known after a failure are not reexecuted. It is possible, when concurrent update is being used, for a logical write W to be performed at a site that fails before $AtBcast(W)$ is performed. In this case, W is not known after the failure, and is reexecuted. We must show that this does not cause a change in the order of dependent operations.

Let W and O be two operations executed before a failure, and let W write before O. Let F represent the event corresponding to making the failure decision. We then have $W \rightarrow_L O \rightarrow_L F$. If O is known after the failure, the properties of atomic broadcasts and the broadcast ordering and blocking rules ensure that W is also known after F. Hence, neither operation is reexecuted. If O is not known but W is, then only O is reexecuted. Since $W \rightarrow_L F$ and the reexecution of O is scheduled after the failure decision, it follows that $W \rightarrow_L O$. If neither operation is known after the failure, the normal recovery mechanism ensures that they are both reexecuted in the correct order. Thus their relative order is preserved in all cases.

We have shown that concurrent update can be used in conjunction with a roll-forward recovery scheme. Roll-forward recovery is inherently expensive and may not be viable without techniques like concurrent update. Although we have presented a particular roll-forward ,recovery scheme, concurrent update can be used with any roll-forward recovery scheme that satisfies the following properties:

(1) Recovery takes place from a previously saved state.
(2) Operations completed before the failure are not reexecuted (or, if they are, the recovery scheme ensures that they have the same effects as the earlier execution).
(3) Failure decisions are serialized with respect to other operations.

9. IMPLEMENTATION EXPERIENCE

This work is a consequence of research on the *ISIS* project at Cornell [3], which aims to develop a system that aids in the automatic construction of fault tolerant distributed software by replicating information. A prototype of the system has been implemented on a cluster of *SUN* workstations, connected by a 10-Mbit Ethernet. A simplified version of the method described in this paper was implemented in *ISIS*, and has been operational since January 1985. One constraint placed on the system architecture by the method is that the order in which operations are executed at each site must respect the order in which update instructions were received there. We were able to achieve this behavior without much difficulty. Additionally, it was necessary to design a failure detection and communication layer satisfying the fail-stop assumptions of Section 1.

The implementation differs slightly from the method as presented here. In this paper, the blocking rule was used to delay the execution of an operation if the required update had not arrived. In *ISIS* this can be dispensed with, for the following reason: *ISIS* uses two-phase locking for concurrency control. Write locks are acquired at all operational sites where a data item resides, while read locks are acquired only locally. Locks are released when a transaction commits. Thus if *W* writes before *O*, *O* cannot acquire a lock at a site until the transaction that performed *W* is committed there. Since commit operations are ordered after all writes in a transaction, a commit instruction will not arrive at a site unless all prior updates have also arrived there. Thus *O* is automatically forced to wait for an update that it depends on, and there is no need to piggyback operation-ID's or to have a blocking rule.

The *ISIS* implementation of concurrent update was compared with a synchronous update algorithm. Detailed performance figures are given in [2]; we confine ourselves to a summary here. Two performance measures are of interest. One, the response time for a typical request, measures the time required to return the result of an operation. We considered a fault-tolerant object implemented using *ISIS* and distributed to 3 sites. A request that acquires a replicated write lock updates a replicated data item and then responds to its caller, sends its reply after about 0.6 seconds; additional updates delay the response by 0.1 seconds each (the difference reflects the one-time cost of concurrency control). When *ISIS* is run in a synchronous mode, verifying that each update has actually completed before the coordinator undertakes any subsequent operations, such a computation requires 1.5 seconds, with additional updates requiring 0.5 seconds each. Moreover, the performance of the synchronous version degrades as the number of sites increases, while the concurrent version gives the same performance regardless of the number of participating sites. Thus, the concurrent update technique can have a substantial impact on performance.

If a high level of concurrency is achieved, a computation can remain active at remote sites long after replying to the process that initiated it. A second performance measure can hence be studied: the total elapsed time between the issuing of a request and the termination of an operation at all sites. In *ISIS*, we find that the single update described above terminates after about 1 second, with subsequent updates delaying termination by about 0.3 seconds each, and with linear degradation as the number of sites increases. It is important to recognize that the delay to termination would not normally be an issue unless computations at different sites compete for a lock, which does not occur frequently. Thus, for *ISIS* and for many other applications, concurrent update permits extremely good performance—almost as good as for a nondistributed system performing the same operations—but with the advantages of replicated data and fault-tolerant execution.

10. RELATED AND FUTURE WORK

In this paper, we have presented the concurrent update method with respect to replicated database systems, where the operations are reads and writes. We have further assumed that a read-one-copy, write-all-copies rule is being followed. In [11], these assumptions are relaxed, and the method is extended to object-based replicated systems, where the operations may be arbitrary operations on objects. It is shown how concurrent update, used in this context, results in highly efficient implementations of replicated objects.

We have found concurrent update so versatile and useful in the *ISIS* system that we are reimplementing the communication subsystem to include a broadcast primitive that performs concurrent update. A detailed description is given in [5]. We believe that a communication sub-system along those lines gives rise to considerable simplicity and efficiency in the design and implementation of replicated fault-tolerant software systems.

Concurrent update reduces the latency arising from replicating data, hence the performance of the concurrency control algorithm could become a critical factor in some systems. In particular, if two-phase locking (with write locks obtained at all copies) were used, and the granularity of locking is individual data items, write locking would require a message to each copy to request a lock and a delay until locks are granted. As a result, concurrent update might not significantly improve system performance. In [7], we present a concurrency control algorithm that exploits available concurrency to a greater degree than conventional methods. This method is not presently being used in *ISIS*, but we expect methods such as this to further improve performance.

Decoupling the transmission of updates from synchronization messages gives rise to a degree of freedom for optimizing performance that is not available in current systems. A message scheduler can be used to delay update instructions until network load diminishes, or to bundle more than one update into one message. Delaying updates, however, could cause operations to block unnecessarily. Performing a broadcast after every write distributes information as soon as possible, but increases message traffic. The optimal degree of delay depends on the relative costs of message transmission and response time, on the characteristics of the message transmission system, and on the degree to which transactions interact with one another. Constructing a message scheduler that balances these factors is an area for future research.

The concurrent update method respects the order $<_T$ on the operations of a transaction T. It is possible to perform transformations on the specification of a transaction, reordering these operations in a way that might lead to more efficient transmission of updates. We are investigating the use of methods from data flow analysis in this connection [9].

11. CONCLUSIONS

We have described a technique to reduce the overhead introduced when updates have to be made on data that are replicated in a distributed system. By relaxing the level of synchronization employed to maintain the consistency of replicated data, the time required to carry out operations is decreased. At the same time, the method does not violate the consistency of the data. We have presented the technique with respect to replicated database systems, but it can be applied to asynchronous distributed systems in which data are replicated for availability or fault tolerance. It demonstrates that such replication can be supported without additional latency when operations are performed. In fact, if a transaction is executed at a site holding a copy of all the data items on which it operates, concurrent update might allow it to execute as fast as if replication were not in use. Replicated systems are inherently expensive, and techniques such as this are necessary if they are to provide adequate performance.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions of Dale Skeen and Fred Schneider, who read earlier versions of this paper and provided helpful suggestions. Our thanks also go to the referees, whose many detailed comments have greatly improved the presentation of this work.

REFERENCES

1. BERNSTEIN P. AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–222.
2. BERNSTEIN P. AND GOODMAN, N. The failure and recovery problem for distributed databases. In *Proceedings of the 2nd Symposium on Principles of Distributed Computing* (Montreal, Canada, Aug.), ACM, New York 1983, 114–122.
3. BIRMAN, K. Replication and fault-tolerance in the *ISIS* system. In *The 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4) 1985. To be published.
4. BIRMAN, K., DIETRICH, W., EL ABBADI, A., JOSEPH, T. AND RAEUCHLE, R. An overview of the ISIS project. Tech. Rep. TR 84–642, Department of Computer Science, Cornell University, Ithaca, NY. Oct. 1984.
5. BIRMAN, K., JOSEPH, T., AND SKEEN, D. Reliable communication in an unreliable environment. Tech. Rep. TR 85–694. Department of Computer Science, Cornell University, Ithaca, NY. Aug. 1985.

6. Birman, K., Joseph, T., Raeuchle, R., and El Abbadi, A. Implementing Fault-Tolerant Distributed Objects. *IEEE Trans. Softw. Eng. 11*, 6 (June 1985) 502-508.
7. Birman, K., Joseph, T. and Raeuchle, T. Extending resilient object types efficiently. The *2nd GI/NTG GMI Conference on Fault-Tolerant Computing Systems.* (Bonn, West Germany), Sept. 1984. Springer-Verlag.
8. Chang, J. and Maxemchuk, N. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 251-273.
9. Fisher, J. A., Ellis, J. R., Ruttenberg, J. and Nicolau, A. Parallel processing: A smart compiler for a dumb machine, Tech. Rep. Department of Computer Science, Yale University, June 1984.
10. Gray, J. Notes on Database Operating Systems. *Lecture Notes in Computer Science 60*, Springer-Verlag, New York 1978.
11. Joseph, T. Low cost management of replicated data. Department of Computer Science, Cornell University, Ph.D. dissertation. Jan. 1986.
12. Kohler, W. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comput. Surv. 13*, 2 (June 1981), 149-184.
13. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 95-114.
14. Papadimitriou, C. H. The serializability of concurrent database updates. *J. ACM 26*, (Oct. 1979), 631-653.
15. Schlichting, R. D. and Schneider, F. B. Fail-Stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (Aug. 1983), 222-238.
16. Schneider, F., Gries, D. and Schlicting, R. "Fault-tolerant broadcast. *Sci. Comput. Program. 4*, 1984. 1-15.
17. Skeen, D. Determining the last process to fail. *ACM Trans. Comput. Syst. 3*, (Feb. 1985), 15-30.

Received November 1984; revised August 1985; accepted September 1985

178

Enclosed is a copy of a technical report produced by the ISIS group. This report was produced under contract number NAG2-593.

Respectfully yours,

Susan Allen,
ISIS Project Secretary
(607) 255-9198
