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

Kenneth P. Birman
Thomas A. Joseph

TR 84-644
October 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

LOW COST MANAGEMENT OF REPLICATED DATA IN FAULT-TOLERANT DISTRIBUTED SYSTEMS

Kenneth P. Birman and Thomas A. Joseph

Department of Computer Science
Cornell University

ABSTRACT

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 ensure that the copies of replicated data are kept consistent introduces a delay when operations are performed. In this paper, we describe a technique that relaxes the usual degree of synchronization, permitting copies of replicated data 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 roll-back and roll-forward failure recovery mechanisms.

This work was supported in part by a grant from the Sperry Corporation.

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; hence fault tolerance can 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 the replicated information. The associated overhead manifests itself in 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 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* [Sch83]: the only failure they suffer is a halting failure - they stop processing, and the other sites are informed of their failure[1]. We require the following property of the communication medium: if a site A sends a number of messages to a site B, B receives them in the same order that A sent them. We also assume that the bandwidth of the communication medium is sufficiently high that overloading does not occur. Although we describe the method with respect to replicated databases, it is applicable to more general replicated systems that satisfy the properties described above.

In the next section, we describe our model for replicated systems. It is similar to the model for replicated databases described in [B&G-83]. In section 3, we examine the concept of "order"

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

as applied to an asynchronous system, and draw some conclusions about the message patterns in such systems. Section 4 describes 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 modified in Section 6 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, and sections 9 and 10 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 processing sites. The operations that can be performed on a replicated data item are *read*, which returns its value, and *write*, which changes its value. 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, while a logical write results in a physical write of all the copies.

High level operations like an *insert* in a queue are modeled by *transactions*. A transaction $T = (O_T, <_T)$ is a set of logical operations O_T and a partial order $<_T$ on them. A logical read (write) 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. (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).) The partial order $<_T$ embodies the data flow relationships between the operations in O_T, and a correct execution of a transaction must observe this order.

An *execution log* $E = (O_E, <_E)$ for a concurrent execution of transactions $T_0, T_1, \cdots, T_n$ is a set O_E, which is the union of O_{T_i} for all i, and a partial order $<_E$ on the operations in O_E. An execution log represents the order in which logical operations are executed by the system. If

$o_i <_E o_j$, we say that o_j is *scheduled after* o_i. The mechanisms used to enforce this order 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.

An execution log E is *correct* if it respects the partial order specified by each transaction, i.e. $\forall i : <_E \supseteq <_{T_i}$, and if E is *serializable*. E is serializable if there exists a total order $<_S$, called a *serialization order*, on the set $\{T_0, T_1, \cdots, T_n\}$ such that if an operation o_i in transaction T_i *conflicts* with an operation o_j in T_j, and if $T_i <_S T_j$, then $o_i <_E o_j$. Two logical operations conflict if they both operate on the same data item, and at least one of them is a write.

The execution system uses a *concurrency control algorithm* to ensure that all execution logs are correct. A number of such algorithms are known, and are discussed in [Pap79, B&G-81, B&G-83, Koh81]. Concurrency control algorithms fall into two classes: the *pessimistic* algorithms, which order reads and writes so that a correct execution log is produced, and the *optimistic* algorithms, which permit reads and writes to occur in arbitrary order, but subsequently abort transactions that render the log unserializable. Henceforth, we will assume that a pessimistic algorithm is in use, although our methods can be extended to deal with optimistic algorithms as well. The details of the particular algorithm used are not essential to our treatment.

Given a correct execution log E, we say that a write operation $W(x, T_a, s)$ *writes-before* an operation $O(x, T_b, t)$ (where O = R or W) if $W(x, T_a, s)$ and $O(x, T_b, t)$ belong to O_E, $W(x, T_a, s) <_E O(x, T_b, t)$, and there is no $W(x, T_c, v)$ such that $W(x, T_a, s) <_E W(x, T_c, v) <_E O(x, T_b, t)$. In other words, W writes-before O if O is scheduled after W, O and W conflict, and there is no other conflicting write operation that is scheduled after W but before O. It follows from the definition of serialization order that if $W(x, T_a, s)$ *writes-before* $O(x, T_b, t)$, then $T_a <_S T_b$ in any serialization order $<_S$.

In Figure 1, we show three transactions, T_1, T_2 and T_3, a correct execution log E for these transactions. There are three sites a, b, and c in the system, and the data items x, y, and z each reside at all the sites. The partial order $<_E$ is the transitive closure of the relations shown in the figure. A serialization order S corresponding to E is $T_2 <_S T_3 <_S T_1$. $W(x, T_3, c)$

writes-before $R(x, T_1. a)$, $W(y, T_3, c)$ *writes-before* $R(y, T_1, a)$, and $W(x, T_2, b)$ *writes-before* $W(x, T_3, c)$.

3. Order in an asynchronous system

An execution log depicts a temporal order between events. As Lamport has observed, 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 of 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.

In particular, for a concurrency control algorithm to enforce an order between events at different sites, it must cause *synchronization messages* to be sent between them. A *message log M*

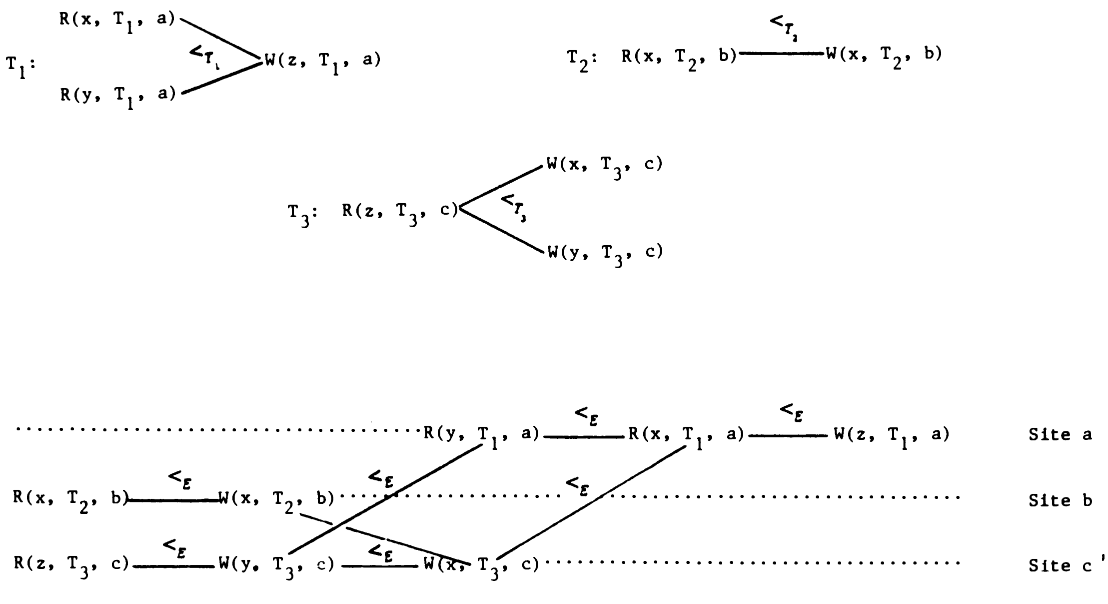

Figure 1: Three transactions and a correct execution log

for a particular execution log E is a record of all logical operations carried out at each site, together with message transmissions between the sites. It consists of a set of *events*, where an event is the execution of a logical operation, the sending of a message, or the receipt of a message, Additionally, it records, for each site, the order in which events are carried out at that site. We use the notation send(m) (receive(m)) for the event 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 message log induces a temporal order $\rightarrow$ on its events. As in [Lam78], this is defined as the smallest relation satisfying the following conditions.

(1) If e_i and e_j are two events at the same site, and e_i occurred before e_j, $e_i \rightarrow e_j$.

(2) If e_i is the sending of a message, and e_j is its receipt, $e_i \rightarrow e_j$.

(3) If $e_i \rightarrow e_j$ and $e_j \rightarrow e_k$, then $e_i \rightarrow e_k$.

An execution log E represents an ordering of a set of logical events. For a physical execution to correctly implement the behavior specified by such a log, it must be the case that $<_E \subseteq \rightarrow$. That is, the logical order of events is reflected in the temporal order of their execution. This leads to the following theorem.

Theorem

For any correct execution log E that contains two operations $W(x, T_a, s)$ and $O(x, T_b, t)$, such that $W(x, T_a, s)$ *writes-before* $O(x, T_a, t)$, the message log contains a sequence of messages $m_0, m_1, \cdots, m_n$ such that sender(m_0) $= s$, $W_j(x, T_b, t) \rightarrow$send($m_0$), receiver($m_i$) $=$ sender(m_{i+1}), receive(m_i) $\rightarrow$ send(m_{i+1}), receiver(m_n) $= t$, and receive(m_n) $\rightarrow R_i(x, T_a, s)$.

Proof

Since $W(x, T_a, s)$ *writes-before* $O(x, T_b, t)$, $W(x, T_a, s) <_E O(x, T_b, t)$. Because $<_E \subseteq \rightarrow$, $W(x, T_a, s) \rightarrow O(x, T_b, t)$. The result follows from the definition of $\rightarrow$.

This theorem states that there is always a "path" of synchronization messages leading from a write operation to an operation that depends on the value written. This result will be used in section 4.

4. Delay introduced by replication

If $W(x, T_a, s)$ *writes-before* $O(x, T_b, t)$ in a non-replicated system, the value of x seen by O is always the value written by W (here $s = t$). In a replicated system, however, the O could take place at a site different from the one at which W is issued, and the system must provide the same effect as if x were non-replicated. In other words, a logical write must be implemented *atomically*.

In the simplest implementation of atomic logical writes, the value to be written is broadcast to all sites, 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 next operation can proceed. Figure 2 depicts this latency. In this paper, we present a solution that permits the operation after a write to proceed as soon as possible, with the physical writes being performed concurrently at the other sites, and with the guarantee that a physical write at a site will be completed before the next read of the same item at that site. We do this in two stages. First, an algorithm is developed that piggybacks update messages on messages used by the concurrency control algorithm for synchronization. Then, a modification of the algorithm is

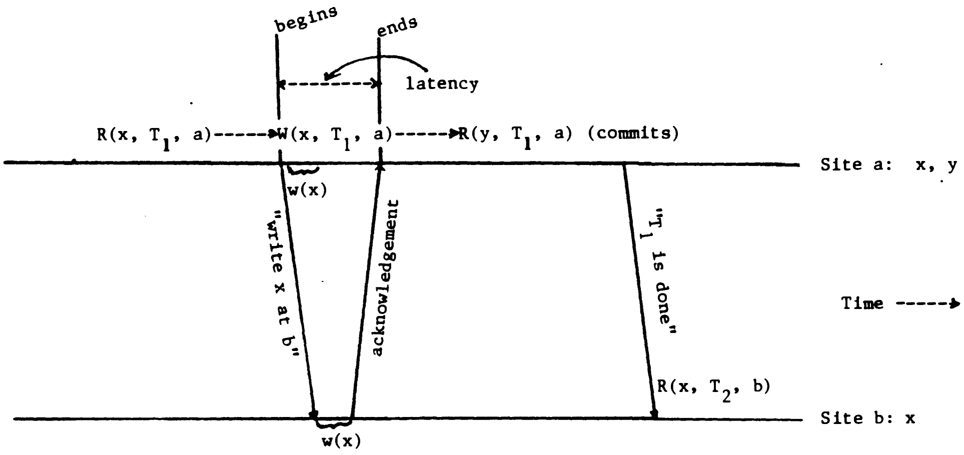

Figure 2: Latency resulting from a replicated update

introduced that permits updates to be broadcast concurrently, yielding the desired result.

5. Piggybacked Update

It is clear that if $W(x, T_a, s)$ *writes-before* $O(x, T_b, t)$, correctness is not violated by delaying T_a's physical write at t, provided that it is carried out before T_b actually performs O. Consider the *update message* that is sent from site s to site t, where a physical write on x must occur. The theorem in section 2 shows that there is always a sequence of synchronization messages upon which this update message 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 data to be physically written; all this information can be piggybacked upon messages required for concurrency control.

This motivates the algorithm shown in Figure 3. Update messages resulting from a logical write are piggybacked on *all* outgoing synchronization messages. By the theorem, a copy of each message will arrive at its destination before the execution of any operation scheduled after it. The algorithm requires that each site s maintain a buffer $Outgoing_s$ of outgoing update messages, each of which is timestamped uniquely by its sender using a local clock. $Outgoing_s$ contains update messages from site s to other sites, as well as messages that pass through site s en route to other sites. An array $LastSeen_s$ records the timestamp on the last message that s received from every other site. A two-dimensional array $TheirView_s$ records, for each other site t, the value of $LastSeen_t$, as known by site s. It reflects the value of $LastSeen_t$ at some time in the past.

This algorithm sends update messages along all possible paths, and discards them only if a copy has already been received by the destination. It has the obvious disadvantage that update messages are piggybacked on messages following paths that might never lead to the required destination, hence requiring unnecessarily large messages and buffers. Some of this overhead can be eliminated if the concurrency control algorithm could indicate which data items a particular synchronization message refers to. For example, with a lock based concurrency control algorithm, the data items corresponding to a particular lock acquisition or release message are always known.

/* Piggybacked update algorithm as followed by site s */

Whenever a synchronization message is being sent to site t

- Piggyback all messages in $Outgoing_s$.
- Piggyback $LastSeen_s$ and $TheirView_s$.

When a synchronization message is received from site t

- For each site x, accept all piggybacked update messages sent from x whose timestamps are greater than $LastSeen_s[x]$, and set the value of $LastSeen_s[x]$ to the largest such timestamp.
- Set the value of $TheirView_s[t]$ to the piggybacked value of $LastSeen_t$.
- Update the value of $TheirView_s$ using the piggybacked value of $TheirView_t$. For each entry choose the larger of the timestamp already known by s and the timestamp known by t.
- Delete from $Outgoing_s$ all messages from site x to site y with timestamps $\geq TheirView_s[y][x]$.

Figure 3: Piggybacked update algorithm.

In this case, update messages for a data item could be piggybacked only on those synchronization messages that refer to it. It is also possible to periodically flush the contents of $Outgoing_s$ to neighboring sites; this keeps the size of the buffers small. Another obvious optimization is to not piggyback on a message to site t those messages in $Outgoing_s$ that have already been piggybacked on an earlier message to t.

In Figure 4, we show how the latency depicted in Figure 2 is reduced by piggybacked update. In many distributed systems the *number* of messages sent, and not their size, is an overriding cost factor. For example, this is the case if messages are processed by a large number of software layers. In such a system, piggybacking would improve performance. However, as seen in the figure, piggybacking update can cause a large number of updates to be delivered to a site just before an operation that depends on them is to be performed. As a result, the operation has to be delayed until the physical writes are actually carried out. This latency, while usually lower than that caused by waiting for response messages, can be avoided using *concurrent update* which

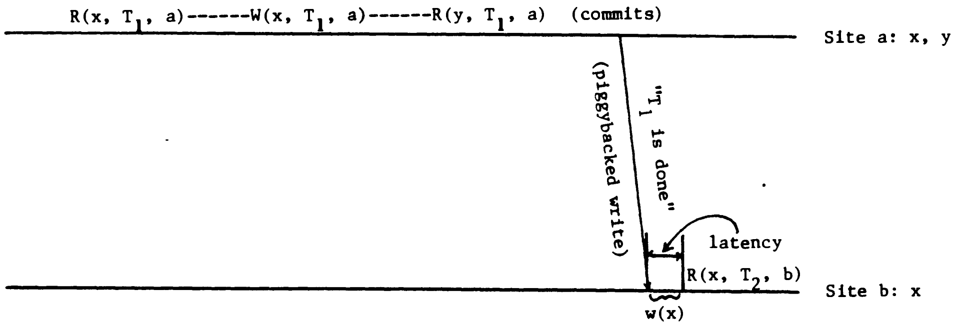

Figure 4: Reduction in latency using piggybacked update.

employs the piggybacked update algorithm as its basis.

6. Concurrent update

The concurrent update scheme requires that each site assign a local timestamp to each logical write operation. Timestamps have the property that if o_i and o_j are two operations at the same site, and if $o_i <_E o_j$, then the timestamp of o_j is strictly greater than that of o_i. The timestamp of a logical write, with the site name appended, is called its *operation-ID*. When a logical write W is performed, its operation-ID is piggybacked as above, instead of the actual update. Update messages are later broadcast to the sites where the copies reside. A transaction does not wait for update messages to be acknowledged, but executes the next operation in parallel with any ongoing updates. The piggybacked operation-ID is guaranteed to arrive at its destinations before the execution of any operation O such that W *writes-before* O. If the corresponding update message has already been received, and the physical write performed, O proceeds without delay. If not, it is blocked until the update message arrives.

The above scheme ensures that an operation does not proceed if a physical write upon which it depends has not been performed. To preserve correctness, an additional constraint must be observed. Delayed updates to the same variable must be carried out in the same order that the logical writes were performed. We now present a broadcast discipline that enforces this condition.

The broadcast discipline is based on an *atomic broadcast* primitive. An atomic broadcast has the following properties.

(1) If two atomic broadcasts are initiated from the same site and if they have one or more destinations in common, the data broadcast are received at all the overlapping destinations in the same order that the broadcasts were initiated.

(2) In the presence of failures, the data broadcast are either received at all the destinations or at none of them.

An atomic broadcast having these properties is easily implemented using a 2-phase commit protocol; this and related protocols are discussed in [Sch84, Cha84].

In our scheme, update messages are always broadcast atomically. The atomic broadcast corresponding to a write W is modeled as an event, $AtBcast(W)$, in the message log. Atomic broadcasts from a site s respect the *broadcast ordering constraint:* If W_1 and W_2 are two conflicting writes at s, and $W_1 <_E W_2$, then $AtBcast(W_1) \rightarrow AtBcast(W_2)$. That is, the atomic broadcast for W_1 is initiated before that for W_2. Physical updates are performed at the time when update messages are received. Of course, it is always possible, as an optimization, to "pack" more than one atomic broadcast to the same set of destinations into one, and to "unpack" them in order.

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.

We have shown how to convert a system that requires waiting for updates to be acknowledged into one where they are done concurrently with other operations. Since only operation-ID's are piggybacked, it requires only little additional message overhead. Moreover, each site needs to retain only one operation-ID for each other site - the one with the largest timestamp. The fact that local timestamps are monotonically increasing and the broadcast ordering constraint ensure that if a site s has received an update message from a site t, then it must have already received all update messages from t with earlier timestamps. Thus, the other operation-

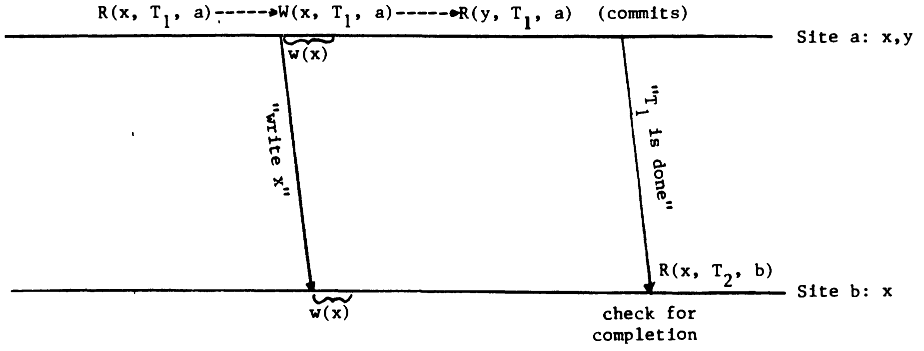

Figure 5: Concurrent update.

id's can be discarded.

7. Commits, aborts and failures

In most database systems, a transaction terminates by executing a *commit* or a *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 permanent, 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 are included in our model by treating them as a write operation on permanent copies of the variables written by the 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 or write 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. When a failure occurs at a site, all actions carried out there (e.g. physical writes, acquisition of locks, etc.) by an uncommitted transaction are not available to the rest of the system. Consequently, many concurrency control algorithms (e.g. available copies [B&G-83]) handle failures by aborting any uncommitted transaction that has executed a logical operation at a site that subsequently fails. When a site is informed of the failure of another, the concurrency control algorithm at that site generates messages to abort such transactions. The concurrent update scheme described above requires a minor modification. It is now possible for a delayed update to arrive at a site after the transaction that performed the update has been aborted. The transaction identifier can be used to detect such messages and ignore them.

If a commit does not have any irreversible side effects (like dispensing money from a machine), commit and abort messages can be transmitted using concurrent update. Then if a commit is performed locally and the site fails before the atomic broadcast is performed, local information is lost and the transaction is aborted at the other sites. The effect is as if a failure occurred during the commit protocol, leading to an abort. On the other hand, if a local commit could have side effects, it is necessary that all sites commit. Concurrent update cannot be used for such commit messages. In this case, the buffer is *flushed*, i.e. all the messages in the buffer are broadcast atomically, and a two or three phase commit protocol is carried out. In general, the buffer must be flushed before performing any operation that has irreversible side effects.

8. Roll-forward recovery

An alternative to recovery from failures by aborting transactions is a roll-forward scheme. Here we outline one such scheme, described in [ISIS-b], and discuss how concurrent update can be used in conjunction with it. In this scheme, a transaction is executed at more than one site, one of which, the *coordinator*, supervises its execution. The other sites are called its *cohorts*. The coordinator periodically takes checkpoints at its cohorts, viz. it transmits its local state to them. When a site fails, the operational sites execute a failure protocol, during which they all agree on the occurrence of, and a relative time for, the failure. This protocol is similar to the *exclude* transaction described in [B&G-83], and has the effect of serializing the failure decision with respect to other operations. Thus a failure decision appears in an execution log just like an operation on a data item. It is scheduled after operations that have been completed before the failure, and all subsequent operations are scheduled after it. New coordinators are then chosen for the transactions coordinated by the failed site, and they resume execution from their last checkpoint. 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 waits until a site has recovered [Ske82]. We assume that the degree of replication of data is such that a copy is always available after a failure, if the failure is not total.

When a coordinator fails, it could have executed operations after it established its last checkpoint. The recovery scheme uses two mechanisms to ensure that the reexecution of these operations by a cohort does not violate correctness. First, operations are executed atomically, i.e. despite failures, they either occur completely, or not at all. Second, the results of all completed operations are *retained.* If a cohort reissues a completed operation, the operation is not actually carried out again; instead the cohort receives the retained results. This is sufficient to guarantee correctness provided that the transaction serialization order used after the failure is consistent with the one used before it. Otherwise, there is no guarantee that a transaction will execute the same sequence of operations after the failure, and operations completed before the failure could leave the system in an incorrect state.

To ensure that ordering is preserved, information about read and write operations that were completed before a failure must be available to the concurrency control algorithm after the failure. Because write operations are atomic and are performed on all copies, information about them is known after a failure (unless the failure is total). To make information about read operations available, information about each logical read must be broadcast to each site where the data item resides. Thus both logical reads and writes require a broadcast of concurrency control information to all sites where a data item resides[2]. If locking is used, this can be achieved by obtaining both read and write locks on all copies.

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 has a data flow dependency on all operations prior to it, and all

[2]A broadcast is not necessary for every read operation. The necessary concurrency control information can be buffered and then transmitted with concurrency control messages relating to write operations that depend on a read, since this information is not required before such writes are performed. The details of this optimization are beyond the scope of this paper.

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 constraint, a checkpoint is considered to conflict with prior write operations in T.

How does concurrent update affect this recovery mechanism? The broadcasts that accompany read operations and checkpoints, which make it appear prohibitively expensive, can now be done concurrently or piggybacked. Thus a seemingly inefficient mechanism is made viable. On the other hand, concurrent update introduces the following possibility. Two logical operations O_1 and O_2 could be performed at a site, and the broadcast corresponding to O_1 delayed until after that for O_2. A failure could occur after the broadcast for O_2 is performed, but before that for O_1. We say that the O_2 is *known* after the failure, while O_1 is not. We need to show that this behavior does not render the recovery mechanism incorrect. Two potential problems arise.

First, if O_2 were a checkpoint and O_1 was an operation prior to it in the same transaction, the recovery scheme would fail because O_1 would not be reissued, even though it was not carried out at the operational sites. However, the broadcast ordering constraint ensures that if a checkpoint is known after a failure, than any operations that executed prior to the checkpoint will be known as well, so this situation cannot occur.

Second, the recovery mechanism requires that the transaction order followed after a failure is consistent with that followed before, and in the absence of concurrent update ensures that this is true. We need to show that the possibility described above does not cause an incorrect serialization order to be followed.

Let E be an execution log in which a failure decision is taken, and let F be the corresponding event in the log. Consider two transactions T_1 and T_2 that had executed conflicting operations O_1 (resp. O_2) before the failure. Assume that $T_1 <_S T_2$. That is, $O_1 <_E O_2 <_E F$. If O_i $(i = 1,2)$ is after a checkpoint in T_i and it is not known after the failure, it is reexecuted and we let O_i' stand for its reexecution. If O_i is prior to a checkpoint, it is not reissued. If it is known after the failure, its retained results are used. In both cases, it is not reexecuted, and we let O_i' stand for O_i itself. We need to show that O_1' and O_2' are executed in an order consistent with the serialization order $T_1 <_S T_2$. There are four cases to consider.

(1) If neither O_1 nor O_2 is known after the failure, they are both reexecuted, and the normal recovery mechanism ensures that they are executed in correct order.

(2) If both O_1 and O_2 are known after the failure, neither is reexecuted (their retained results are used), and the result follows.

(3) If O_1 is known after the failure, but O_2 is not, the recovery mechanism ensures that O_2 will be reissued. O_1 may or may not be reissued, depending on whether it precedes or succeeds a checkpoint T_1. In either case, it will not be reexecuted because its retained results are available. Now we have $O_1 <_E F$ and $F <_E O_2'$, because the reexecution of O_2 is scheduled after the failure decision. This gives us the required result.

(4) It is not possible for O_2 to be known after the failure if O_1 is not. If O_1 and O_2 were executed at the same site, then since $O_1 <_E O_2$, $AtBcast(O_1)$ was initiated before $AtBcast(O_2)$. Property 1 of the atomic broadcast implies that if O_2 is known after the failure, O_1 must be too. If they were executed at different sites, than since O_1 and O_2 conflict, O_2 would have been delayed until the broadcast corresponding to O_1 was received, hence receipt of the broadcast occurred at some site. Property 2 of the atomic broadcast states that if a broadcast is received anywhere, it is received everywhere. Thus, O_1 is known after the failure.

We have shown that it is possible to convert a roll-forward recovery scheme that is correct when each broadcast entails waiting for acknowledgements into one that uses concurrent update. Delaying updates does not invalidate the recovery scheme. Roll-forward recovery is an expensive technique and may not be viable without 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 executed before the failure are not re-executed (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. Related and future work

This work is a consequence of research on the *ISIS* project at Cornell [ISIS-a], which aims to develop a system that aids in the automatic construction of fault tolerant distributed software by replicating information. The method described in this paper is being implemented in the *ISIS* system. One constraint placed on the system architecture by the method is that the order in which conflicting operations are executed must respect the order in which they were received by each site. We have been 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. Concurrent update reduces the latency arising from replicating data, hence the performance of the concurrency control algorithm became critical. 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 [ISIS-c], we present a concurrency control algorithm that exploits available concurrency to a greater degree than conventional methods. Using this method together with concurrent update, very good performance will be possible in fault-tolerant programs.

Concurrent update gives us a degree of freedom for optimizing performance that is not available in current systems. A message scheduler can be used to delay update messages until network load diminishes, or to bundle more than one message into one transmission. On the other hand, delaying messages could unnecessarily cause operations to block. Performing a broadcast for 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.

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 update messages. We are investigating the use of methods from data flow analysis in this connection [Fis84], and initial results have been promising.

10. Conclusions

We have described a technique to reduce the overhead introduced when data 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, provided that the bandwidth of the communication medium is not a bottleneck. In fact, if a transaction is executed at a site holding copies 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.

11. Acknowledgements We would like to acknowledge the contributions of Dale Skeen and Fred Schneider, who read earlier versions of this paper and provided helpful suggestions.

REFERENCES

[Abb84] El Abbadi, A., Skeen, D. *et al.* An efficient, fault-tolerant algorithm for replicated data management. *Forthcoming.*

[B&G-81] Bernstein, P., Goodman, N. Concurrency control in distributed database systems. *ACM Computing Surveys* **13**, 2 (June 1981), 185-222.

[B&G-83] Bernstein, P., Goodman, N. The failure and recovery problem for distributed databases. *Proc. 2nd PODC,* Montreal, Canada, Aug. 1983, 114-122.

[Cha84] Chang, J., Maxemchuk, N. Reliable broadcast protocols. *TOCS* 2, 3 (Aug. 1984), 251-273.

[ISIS-a] Birman, K., Dietrich, W., El Abbadi, A., Joseph, T., Raeuchle, R. An overview of the ISIS project. Department of Computer Science, Cornell University, Tech. Rep. TR-84-62?, (Oct 1984).

[ISIS-b] Birman, K., Joseph, T., *et al.* A technique for implementing fault-tolerant objects. Department of Computer Science, Cornell University, Tech. Rep. TR-84-594, (March 1984). To appear, *4th Symp. on Reliability in Dist. Software and Database Systems,* Oct, 1984.

[ISIS-c] Birman, K., Joseph, T., Raeuchle, T. Implementing Resilient Objects Efficiently. *2nd GI/NTG/GMR Conference on Fault-Tolerant Computing Systems,* Oct. 1984, also available as Department of Computer Science, Cornell University, Tech. Rep. TR-84-622, (June 1984).

[Fis84] Fisher,J.A., Ellis, J.R., Ruttenberg J.and Nicolau, A. Parallel Processing: A smart Compiler for a Dumb Machine. Department of Computer Science, Yale University, June 1984.

[Koh81] Kohler, W. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys* **13**, 2 (June 1981), 149-184.

[Lam78] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM* **21**, 7 (July 1978), 95-114.

[Pap79] Papadimitriou, C.H., The serializability of concurrent database updates. *JACM* **26**, 4 (Oct. 1979), 631-653.

[Sch83] Schlichting, R.D., Schneider, F.B. Fail-Stop processors: An approach to designing fault-tolerant computing systems. *Trans. Computer Systems* **1**, 3 (Aug. 1983), 222-238.

[Sch84] Schneider, F., Gries, D., Schlicting, R. Fault-tolerant broadcast. *Science of Computer Programming* 4 (1984), 1-15.

[Ske82] Skeen, D. Crash recovery in a distributed database system. (Ph.D. Thesis), ERL Memo M82/45, University of California, Berkeley, May 1982. (Also to appear in: Lecture Notes in Computer Science, Springer-Verlag).