Exploiting Atomic Broadcast in Replicated Databases (Extended Abstract)

Divyakant Agrawal[1], Gustavo Alonso[2], Amr El Abbadi[1], and Ioana Stanoi[1]

[1] Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

[2] Institute for Information Systems, Database Group, ETH Zentrum, CH-8092 Zurich, Switzerland

1 Introduction

In spite of the fact that many applications require replicated databases either for performance or fault-tolerance, replication has remained a research issue until recently. Today, the demand for practical replication schemes has greatly increased and some simple protocols are being implemented in databases (Oracle and Sybase, for instance) or in application development tools (Lotus Notes). These, however, are ad hoc implementations and the issue of replicated data management is still a source of controversy among database practitioners and researchers. On one hand, traditional synchronous protocols are too expensive in terms of message cost and communication latency, and they are susceptible to deadlocks when compared to non-replicated databases. An alternative approach based on asynchronous updates may result in inconsistencies and an ever increasing number of reconciliation rules are needed [5].

The distributed systems and computing communities have in general been interested in the broader problem of fault-tolerance in distributed applications. In particular, several systems such as ISIS [4], Amoeba [8], Trans/Total [10], and Transis [2] provide broadcast communication to support fault-tolerant applications. Broadcast communication primitives typically provide reliability, atomicity, and ordering properties at a single operation (or equivalently message) level. Transactions, on the other hand, require reliability, atomicity, and ordering guarantees not for a single operation but for a group of operations. In order to use broadcast for transaction management in replicated data, this mismatch needs to be addressed [12].

In this paper, we propose a series of protocols that bridge the gap between database transactions and broadcast communication in the context of replicated databases. Our goal is twofold. From a database perspective, by using a powerful broadcast communication primitive, we hope to simplify the management of replicated databases and obtain some benefits. In particular, broadcasts can be used to guarantee the consistency of multiple copies of data and at the same time reduce the probability of deadlocks. In fact, in this paper we show that broadcast primitives hold the promise of eliminating single object deadlocks, and either of localizing deadlock resolution or even completely eliminating deadlocks. Our second goal is to provide an application platform for evaluating some of the broadcast primitives. We show that if broadcasts are powerful enough to provide certain "atomicity" guarantees, the database application can execute transactions very efficiently while almost completely eliminating deadlocks. However, if such atomic properties are weakened due to various system assumptions, a database application that requires strict database consistency must perform its own atomic commitment operations in spite of the properties of broadcast primitives.

2 System and Communication Model

A *distributed database* [3] consists of a set of *objects* stored at different sites connected by a communication network. Users interact with the database by invoking *transactions*. A transaction is a sequence of read and write operations that are executed atomically, i.e., a transaction either *commits* or *aborts* the results of all its operations. A commonly accepted correctness criteria in databases is the *serializable* execution of transactions, which is enforced locally by strict two phase locking. We assume a fully replicated database in which the multiple copies of an object must appear as a single logical object. This is termed as *one-copy equivalence* and is enforced by a *replica control* protocol. The correctness criterion for replicated databases is *one-copy serializability*, which ensures both one-copy equivalence and the serializable execution of transactions. The *atomic broadcast* used in this paper is assumed to have the following properties [7]:

1. If a correct (non-failed) site broadcasts a message m, the primitive ensures that the message will be delivered to all operational sites. Furthermore, if a site delivers a message m, then all operational sites deliver m.
2. A message is delivered at most once, and only if it was actually broadcast.
3. If sites p and q deliver broadcast messages m and m', then m and m' are delivered in the same order at all sites.

In the Section 3, we assume no site failures, however, in the rest of the paper we consider sites that are *fail-stop* [13]. In [1] issues involving failures and recovery are presented in more detail.

3 A Naive Broadcast Based Replica Control Protocol

We first develop a simple protocol for maintaining a replicated database in an idealized environment when there are no failures.

Given that the underlying communication system supports totally ordered atomic broadcasts, the state machine approach [14] can be used to maintain replicated data. In the state machine approach operations are processed one at a time at every site in the same order. We can adopt this approach by requiring that a transaction initiated at a site broadcasts all its operations to all other sites using the atomic broadcast and conflicting operations are executed in the order they are received. More formally, a transaction T_i executes as follows:

1. Every read or write operation on an object is executed by broadcasting the operation to all sites. For operation execution, the transaction waits until the operation has been delivered and executes locally before broadcasting the next operation.
2. Operation execution at every site follows the strict two-phase locking rule. Read locks are obtained at all sites but the read operation is performed only at the initiator. Write locks are obtained and writes are performed at all sites.
3. After executing all its operations T_i issues a commit or an abort, which is broadcast to all sites. As a result T_i is committed or aborted at consistently at every site.

The protocol is fairly simple and use of the state machine approach makes the correctness argument a straightforward extension of the serializability argument in a non-replicated databases[3].

Since this protocol uses atomic broadcast and the state machine approach, it is desirable to localize the deadlock resolution at every site. However, it is important for the correctness of the protocol that every site makes the same decisions. An immediate consequence of this requirement is that every site must use the same approach for resolving deadlocks, i.e., all of them use either deadlock detection using wait-for-graphs or deadlock prevention using wound-wait or wound-die [9]. The manner in which commit operations (and hence lock releases) are processed by the lock managers has direct ramifications on the choice of the deadlock resolution mechanism. If the lock manager does not process new operations before the completion of a prior commit operation, either strategy can be used. However, this is not required by the protocol. Hence, different sites may potentially execute o in different states. At site S_A, the commit may have completed (and associated locks are released) and therefore o can be granted a lock whereas at site S_B, the commit may still be in progress and hence o cannot be granted a lock. If deadlock prevention is used, this may result in inconsistent lock decisions at S_A and S_B for operation o; hence, deadlock prevention cannot be used in this case. On the other hand, if cycle-based deadlock detection is used, no such inconsistency would arise. Another restriction on the deadlock prevention strategy is that it cannot be based on timers and time-outs since this may result in lock managers making different decisions for the same operation. Finally, the state machine approach precludes the use of multiple threads in the lock manager, at least in their conventional form. Multiple threads can be used only if the order of execution of operations does not differ from the order in which they are delivered.

Next we list and analyze the advantages of using atomic broadcast over replica management protocols based on point-to-point communication:

1. **Elimination of acknowledgements.** Elimination of explicit acknowledgements at the application level becomes possible due to broadcast communication which in turn reduces the communication latencies incurred to execute transactions.

2. **Elimination of global synchronization.** Since all sites make the scheduling decisions locally, they do not require any global synchronization for handling deadlocks.
3. **Elimination of deadlocks involving a single replicated object.** Since concurrent and competing requests for the same object will be delivered to and processed by every lock manager in the same order, transactions could not be involved in single object deadlocks.
4. **Elimination of two-phase commitment.** Since there are no failures and all sites make the same scheduling decisions, the two-phase commit protocol is not needed.

4 Localizing Read Operations

An obvious drawback of the previous protocol is that read operations are performed globally (i.e., broadcasting read lock requests to all database sites), which is an overkill. In general, read operations are significantly more frequent than write operations. Hence, localizing the execution of read operations is desirable and can result in significant performance improvements. The state machine approach ensures that conflict between read and write operations is detected at every site. Clearly to ensure serializability, detection of such a conflict at any one of these sites would have been sufficient.

A simple extension of the state machine approach is to execute read operations locally and write operations globally. This could, however, result in inconsistent decisions at different sites. Since read operations obtain locks at a single site, a write operation may be executed at one site and be blocked on a read lock on another site resulting in different states at different sites. A straightforward way of resolving this is by requiring that the completion of write operations at every site be explicitly acknowledged to the initiator. This approach requires explicit acknowledgements and hence, the advantages of using broadcast primitives are not clear. Furthermore, deadlocked transactions may involve multiple sites. The only advantage that remains is the elimination of single object deadlocks.

We now explore a variation of this protocol, referred to as the *broadcast-writes* protocol, that eliminates the need for acknowledgements while maintaining the local execution of read operations. A transaction T_i at site S_A executes as follows:

1. A read operation is executed locally after obtaining the corresponding read lock.
2. A write operation is broadcast to all database sites and executed only after delivery.
3. When the lock manager at site S receives write operation $w_i[x]$, it checks if the lock can be granted. If granting is successful, the operation is forwarded to the data manager at S. If the lock cannot be granted, x is either locked by readers ($3a$.) or a writer ($3b$.).
   (a) For every transaction T_j that holds a read lock on x, the lock manager checks whether T_j had already broadcast a commit. If so, the lock manager blocks $w_i[x]$ until it receives T_j's commit. If S had not sent T_j's

commit, T_j is aborted and T_i is granted the write lock (note that all aborted transactions were initiated at site S). T_j's abort is broadcast to release all locks held by T_j at other sites.
   (b) If x is locked by a writer, $w_i[x]$ is resolved at S using the appropriate deadlock resolution mechanism (which will be the same at all database sites).
4. T_i terminates either by broadcasting a commit operation if atomic broadcast ensures the "all-or-nothing" property (a broadcast message is either delivered to all sites or to none of them regardless of site failures), or T_i terminates by employing an atomic commitment protocol.

The correctness of the proposed protocol is argued in [1]. In this protocol, deadlocks only involve write operations. This is because a write operation is never blocked indefinitely on account of a read operation. Since write operations are performed globally, a cycle involving write operations will be detected at every site in the system. Hence, deadlocks can be detected and resolved consistently at all sites locally. Thus, localizing read operations allows us to realize the benefits of using atomic broadcasts as mentioned before.

5 Localizing Transaction Execution

The broadcast-writes protocol has several advantages over traditional protocols. However, it is not radically different from protocols based on point-to-point communication. In this section, we explore the possibility of completely localizing transaction execution. This is achieved by deferring update operations until commit time, when a single message with all updates is sent to all other sites (a similar technique, *field-calls*, has been suggested to minimize the time interval during which a data item must be locked to be updated [6]). The advantage of this approach will be that either only two broadcast messages are needed per transaction or the cost of atomic commitment depending upon whether the "all-or-nothing" property is supported by the communication subsystem. Since there are only two broadcast operations involved, the communication overhead is significantly reduced. A transaction T_i executes as follows:

1. A read operation $r_i[x]$ is executed locally by obtaining a read lock on x.
2. A write operation $w_i[y]$ is deferred until T_i is ready to commit.
3. To terminate, T_i broadcasts its deferred writes $w_i[x_1, \ldots, x_n]$ to all sites. On receiving $w_i[x_1, \ldots, x_n]$, the lock manager on site S grants all write locks to T_i atomically as in the broadcast-writes protocol, and then the writes are executed at S.
4. After all the writes of T_i are executed locally, T_i broadcasts its commit operation to all sites. T_i terminates after the delivery and execution of its commit locally.
   If the broadcast communication does not support the "all-or-nothing" property then step 4 can be incorporated into an atomic commitment protocol.

This protocol is also based on reading one copy and writing all copies of replicated objects. Its correctness is a direct consequence of the correctness of the broadcast-writes protocol. However, the similarity ends here. The above protocol exploits the benefits of atomic broadcasts significantly. Unlike before, where deadlocks could involve write operations of different transactions, here deadlocks are not possible. This is because all the write locks for a transaction are obtained in a single atomic step at the local lock manager. Conflicts with read locks are dealt with by aborting read operations. Eliminating deadlocks is a significant benefit for replicated databases as has been argued recently by Gray et al. [5].

Since all write operations are known to all sites and they will be eventually executed, the question arises if the extra broadcast at step 4 is superfluous. We illustrate the necessity of step 4 with the help of the following example. Consider a site S_1, where transaction T_1 executes $r_1[x]$ and broadcasts its commit by sending $w_1[y]$. Before $w_1[y]$ is delivered a commitment of transaction T_2 containing $w_2[x, y]$ is delivered at S_1. S_1 can commit both transactions only if it reorders the delivery order of the write operations of the two transactions. Since all other sites are unaware of $r_1[x]$ they will execute the write operations of T_1 and T_2 in the order they are delivered. The addition of step 4 to the protocol avoids this inconsistency.

We now develop a protocol that overcomes the problem described above, and requires a single broadcast per transaction. A version number is maintained with each item in the database. A transaction T_i executes according to the following rules:

1. A read operation $r_i[x]$ is executed locally by obtaining a read lock on x.
2. A write operation $w_i[x]$ is deferred until T_i is ready to commit.
3. When the site that initiated T_i is ready to commit, if T_i is a read-only transaction, the decision to commit is done locally and no message is broadcast. Otherwise the site broadcasts the set of reads with their version numbers and the set of writes.
   (a) On receiving the set of reads and writes of T_i, the lock manager on S first checks if the version of the items read by T_i are obsolete (that is if any of the versions on S are greater than the versions read by T_i). If so, T_i is aborted. Otherwise, S proceeds with the attempt to grant atomically all the write locks.
   (b) If a write lock cannot be granted due to a conflict with a read operation, the reading transaction is aborted and T_i receives the lock. This step is the same as (3a.) of the second protocol.
   (c) Once all write locks are obtained, S executes the write operations and increments the version numbers of each data item.
4. T_i terminates at S as soon as it atomically obtains the write locks and successfully executes its write operations.

Due to the fact that the commit decision is not being broadcast by the site initiating T_i, the proof of correctness is different than the preceding protocol [1]. It is based on the fact that the order of conflicting operations is the same at all sites. This last protocol has the advantage over the previous one that it involves at most one broadcast operation. The decision to commit a writing transaction is taken independently at all sites and a second broadcast is not needed. The issue of maintaining the delivery order of write operations at a lock manager is still pertinent. In particular, if an operation $w_i[x_1, \ldots, x_n]$ is delayed by the lock manager at site S on account of conflicting writes, the question arises as to whether the lock manager can serve the next request in the input queue. A conservative but safe approach would be to delay all processing until $w_i[x_1, \ldots, x_n]$ can be served at S. In other words, this corresponds to processing write operations serially at every site. This can be relaxed by processing write operations that are disjoint with pending write operations in the queue. In other words, the delivery order of conflicting write operations has to be preserved but it is not necessary to do the same for non-conflicting write operations.

6 Discussion

In this paper, we have developed a series of replica management protocols that exploit the properties of atomic broadcasts. Our first protocol uses the state machine approach with atomic broadcast resulting in several advantages. Although the state machine approach simplifies the protocol design, it suffers from excessive broadcast overhead since every read operation is broadcast to every site. We therefore developed a protocol with localized reads to retain the advantages of using broadcasts. In the final refinement, we presented a protocol in which transactions are executed locally and atomic broadcasts are used to terminate the transactions. Depending upon the properties available from the broadcast system, this resulted in transactions incurring a cost of either one or two broadcasts or atomic commitment. In either case, the resulting protocol has a significant advantage since transactions do not incur any communication delays during their execution except at termination.

Our work points out several interesting directions for future research. From a systems point of view, a reevaluation of the broadcast primitives is needed to figure out what properties can be realistically achieved in failure-prone distributed systems, while being powerful enough to provide useful services to the application. From the database point of view, a study of relaxed consistency conditions, e.g., causal serializability [11] may be beneficial within the context of weaker broadcast properties, e.g., based on causal [15] rather than atomic broadcasts.

References

1. D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting Atomic Broadcast in Replicated Databases. Technical report, Department of Computer Science, University of California at Santa Barbara, 1996.
2. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication sub-system for high availability. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, 1992.

3. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.
4. K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Press, 1994.
5. J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, June 1996.
6. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman, 1993.
7. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcast and Related Problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–147. Addison-Wesley, 1993.
8. M. Frans Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating Systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230, May 1991.
9. E. Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4):303–328, December 1987.
10. L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, 1994.
11. M. Raynal, G. Thia-Kime, and M Ahamad. From Serializable to causal Transactions for Collaborative Applications. Technical report, IRISA, 1996. Publication Interne No. 983.
12. A. Schiper and M. Raynal. From Group Communication to Transactions in Distributed Systems. *Communications of the ACM*, 39(4):84–87, April 1996.
13. R. Schlichting and F. B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1982.
14. F. B. Schneider. Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, April 1982.
15. I. Stanoi, D. Agrawal, and A. El Abbadi. Using Broadcast Primitives in Replicated Databases. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1997.

Acknowledgements

This research was partially supported by LANL under grant number 6863V0016-3A, by CalTrans under grant number 65V250A, and by the NSF under grant numbers IRI94-11330, CDA94-21978 and CCR95-05807.