Using Broadcast Primitives in Replicated Databases [*][†]

I. Stanoi D. Agrawal A. El Abbadi

Dept. of Computer Science
University of California
Santa Barbara, CA 93106
E-mail: {ioana,agrawal,amr}@cs.ucsb.edu

Abstract

We explore the use of different variants of broadcast protocols for managing replicated databases. Starting with the simplest broadcast primitive, the reliable broadcast protocol, we show how it can be used to ensure correct transaction execution. The protocol is simple, and has several advantages, including prevention of deadlocks. However, it requires a two-phase commitment protocol for ensuring correctness. We then develop a second protocol that uses causal broadcast and avoids the overhead of two-phase commit by exploiting the causal delivery properties of the broadcast primitives to implicitly collect the relevant information used in two-phase commit. Finally, we present a protocol that employs atomic broadcast and completely eliminates the need for acknowledgements during transaction commitment.

1 Introduction

Recently there has been increasing interest in the development of broadcast protocols for disseminating information in distributed systems [5, 7, 10, 2]. Several broadcast protocols with varying properties have been proposed and implemented in different distributed systems. These protocols range from simple broadcast protocols that essentially ensure eventual delivery to all sites, to more complex protocols that enforce a specific order on the delivery of messages. In the latter case, the causal broadcast protocol ensures that causally related messages are delivered in that order at all sites, while the atomic broadcast protocol ensures that all messages are delivered in the same order at all sites. Various implementations of these protocols have been proposed for different underlying hardware, including Amoeba [8], ISIS [5], Transis [2], Total [9], and Totem [10]. In general, the weaker the ordering requirements the more efficient the protocol.

An important and often cited application of broadcast protocols is the management of replicated data. Recently, there has been increasing interest in the management of replicated databases, where the unit of activity is a transaction consisting of multiple operations that need to be executed atomically as a unit [12]. Two proposals that explore the transaction semantics are [3, 1]. In both cases, the replication management protocols require, in addition to the broadcast primitives, additional techniques to ensure correct transaction execution. Furthermore, they are based on atomic broadcast primitives, which are both expensive and complex to implement in asynchronous systems that are subject to failures. Raynal et al. [11] have proposed protocols that use broadcast primitives but relax the serializability-based correctness criterion.

In this paper, we explore the use of variants of broadcast protocols for managing replicated databases based on serializability. In particular, we start with the simplest broadcast primitive, the reliable broadcast protocol, and show how it can be used to ensure correct transaction execution. The protocol has several advantages, including prevention of deadlocks and the guaranteed commitment of read-only transactions. However, it requires a two-phase commitment protocol for ensuring correctness. We then develop a second protocol that uses the more expensive causal broadcast and avoids the overhead of two-phase commit by exploiting the causal delivery properties of the broadcast primitives to implicitly collect the relevant information used in two-phase commit. Finally, we

[*]This work has been partially supported by the NSF under grant numbers CCR95-05807 and CCR97-12108.

[†]Copyright 1998 IEEE. Published in the Proceedings of ICDCS'98, May 1998 Amsterdam, The Netherlands. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

present a protocol that employs atomic broadcast and completely eliminates the need for acknowledgments during transaction commitment.

2 System and Communication Model

A distributed system consists of a set of distinct sites that communicate with each other by sending messages over a communication network. A common requirement, which we assume in this paper and is easily enforced, is that communication between a pair of sites is FIFO, i.e., messages are delivered in the order in which they are sent. Users interact with the database by invoking *transactions* i.e. a sequence of read and write operations that are executed atomically. A transaction either *commits* or *aborts* the results of all its operations. A commonly accepted correctness criterion in databases is the *serializable* execution of transactions. Since strict two-phase locking[1] is widely used, we assume in this paper that concurrency control is locally enforced by strict two-phase locking at all database sites. In a distributed *replicated database*, copies of an object may be stored at several sites in the network. Multiple copies of an object must appear as a single logical object to the transactions. This is termed as *one-copy equivalence* and is enforced by a *replica control* protocol [4]. The correctness criterion for replicated databases is *one-copy serializability* [4], which ensures both one-copy equivalence and the serializable execution of transactions. For simplicity, we assume that the database is *fully replicated*, i.e., every site stores a copy of all objects in the databases. Furthermore, we assume that a transaction performs all its read operations before initiating any write operations. Note that this assumption does not reduce the application expressibility of the traditional transaction model. In order to enforce this type of transaction execution, we only need to defer the writes of standard transactions.

Numerous proposals have been made to specify broadcast communication models in distributed systems [5, 7, 2, 10, 6]. Each of these broadcast primitives has subtle differences in its specifications as well as the assumptions it makes regarding the underlying system. However, most of the protocols have the following properties, which can also be used to define the simplest primitive, the *reliable broadcast protocol* [7]:

1. Validity: If a correct process broadcasts a message m, then all correct processes eventually deliver m.
2. Agreement: If a correct process delivers a message m, then all correct processes eventually deliver m.
3. Integrity: For any message m, every correct process delivers m at most once, and only if m was previously broadcast by the sender of m.

[1]In strict two-phase locking, transactions hold all locks until termination.

Two events are *causally related* if they are both executed on the same site, or if one of them is the sending of a message and the other is the delivery of the same message. The causal relationship between all events in the system is the transitive closure of these two conditions. A causal broadcast protocol delivers messages in causal order. *Causal broadcast*, in addition to the three properties of reliable broadcast, guarantees the following property:

> Causal delivery: If the broadcast of a message m causally precedes the broadcast of another message m', then no correct process delivers m' unless it has previously delivered m.

Causal broadcast protocols have been implemented in several systems such as ISIS [5], and Transis [2]. *Atomic broadcast* imposes a total order on the delivery of all messages, i.e., it is a reliable broadcast with the following property:

> Ordered delivery: If sites p and q deliver broadcast messages m and m', then m and m' are delivered in the same order at all sites.

It is worth noting that atomic broadcast ensures same order of delivery at all sites, but does not necessarily ensure that messages are delivered in the same order they are sent. However, due to the FIFO assumption about the communication links, if a process atomically (or for that matter reliably or causally) broadcasts a message m_1 before message m_2 then all processes receive m_1 before m_2. Atomic broadcast has been implemented in Amoeba [8], ISIS [5], Transis [2], Total [9] and Totem [10].

Many of the broadcast primitives incorporate in their implementation various techniques for fault-tolerance including the majority quorum approach. In particular, the communication layer maintains a view of the current system configuration. As site failures and recovery occur, the view is dynamically restructured using the notion of majority quorums. As long as the view has majority membership, the system remains operational [5, 13]. Given a view of the current configuration, we are using the notion of read-one write-all in that context. This avoids duplication of efforts to provide fault tolerance at multiple levels.

3 Reliable Broadcast-Based Protocol

Reliable broadcast is a simple communication primitive that is relatively straightforward to implement when compared to ordered broadcast primitives such as causal or atomic broadcast. We therefore begin our exposition by developing a replicated data management protocol that uses reliable broadcast for all communication. Since reliable broadcast guarantees eventual delivery, one of the motivations for our protocol is to remove the need for explicit acknowledgment after every remote interaction. There are numerous replica control protocols that have been proposed in the database literature, however, all these protocols have been studied in the context of a point-to-point communication environment. In this paper, we start by adapting the read-one write-all protocol for broadcast environments.

We are motivated to use reliable broadcast in order to eliminate the need for explicit acknowledgments for the write operations. We shift the responsibility of ensuring the reliability property of the communication to the communication primitives, and simplify the implementation at the application level. Recall that we assume the following transaction execution model: first all read operations are executed at the initiating site, followed by the set of writes which are broadcast to all the sites. Finally, the commit protocol is executed. Furthermore, since the initiator of a transaction knows that all messages it sends out are guaranteed to be delivered, the initiator can issue consecutive *write* operations in a *non-blocking* manner, i.e, it issues a write operation, and immediately proceeds to the next operation. We next describe the details of the protocol.

Reliable Broadcast-Based Protocol.

1. Read Rule. A read operation $r_i[x]$ of a transaction T_i is executed locally by acquiring a read lock at the initiating site of transaction T_i.

   T_i remains blocked until the read lock on x is granted at the initiator.

2. Write Rule. A write operation $w_i[x]$ of a transaction T_i is executed by reliably broadcasting it to all database sites. On delivery, a site S acquires a write lock on x for T_i. If it is currently held by another transaction, then $w_i[x]$ is delayed at S until the lock on x can be granted.

   T_i proceeds to execute the next operation without waiting for all write locks on x to be granted, including at the initiator.

3. Commit Rule. When the initiating site I decides to commit a transaction T_i, it reliably broadcasts a commit request c_i to all the database sites including itself. On delivery of c_i, a site S (including I) checks if T_i has any pending write operations. If this is the case, S broadcasts a negative acknowledgment to all the database sites. Otherwise, a positive acknowledgment is broadcast to all the database sites. A transaction is aborted at all sites including I if there are any negative acknowledgment to T_i's commit request. On the other hand, if all the database sites acknowledge positively, T_i is committed and all locks (the read locks at I and write locks at all sites) held by T_i are released. Since the votes of all participants are broadcast to every participant, the commit protocol is the decentralized two-phase commit protocol [14].

The above protocol is simple and is in fact a fairly straightforward extension of the traditional read-one write-all protocol that uses point-to-point communication primitives. In fact, it could even be used assuming standard point-to-point communication, where transactions optimistically assume all operations will be delivered, and only at commit time the optimistic assumption is validated. However, the reliability property of the reliable broadcast protocol makes the optimistic assumption more appropriate. The protocol also exploits the reliability properties of the broadcast primitive by not requiring explicit acknowledgments for operations that are executed on remote database sites. In that sense, transactions execute without waiting for global synchronization and instead this is verified at the time of transaction commitment.

Reliable delivery is not the sole concern in a transaction-based system: all conflicting operations must be executed in the same order. Since no order guarantees are ensured by the reliable broadcast primitive, write operations may arrive at different sites in different orders. Any such inconsistencies are detected and eliminated at commit time. The correctness of the protocol follows from the fact that two-phase locking is used locally at each site, and that all write operations are broadcast. Two-phase locking ensures that conflicting transactions are committed in the same order as the execution of their conflicting operations. Hence the serialization graph for their execution must be acyclic. To show that the serialization graph is also a one-copy serialization graph, we need to show that the serialization graph for any execution includes all the relevant write order and read orders on transactions. A write order is ensured because all write operations are broadcast to all sites, and concurrent conflicting writes that are delivered in different order at different sites are eventually aborted by our protocol. Furthermore, the use of two-phase locking and the fact that write operations are executed at all sites ensure the read order. Therefore, all transaction executions are one-copy serializable.

An important side-effect of this approach is that since transactions are not blocked for global synchronization, *deadlocks can be resolved consistently.* This can be observed by noting that a transaction can only be blocked at its initiating site during the read phase. Write operations are broadcast in a non-blocking manner and all sites receive write operations and either execute them or delay them until commitment when the initiating transaction is forced to abort. There are two cases to consider. If the blocked operation is a read operation, then a global deadlock cannot occur, since any other blocked operation must be a write operation initiated at another site, which would be aborted at its commit time. Alternatively, if the blocked operation is a write operation then for a global deadlock to occur, it must be blocked on another write operation. This is because all reads are executed before all writes within a transaction and the completion of all write operations is verified at commit time. Furthermore, since transactions are only blocked during the read phase, they cannot be involved in a local deadlock. Hence, by using reliable broadcast and the decentralized two-phase commit protocol we are able to eliminate the need for explicit acknowledgments for write operations as well as eliminate the problem of deadlocks.

A final interesting property of this protocol is that *read-only transactions are never aborted.*

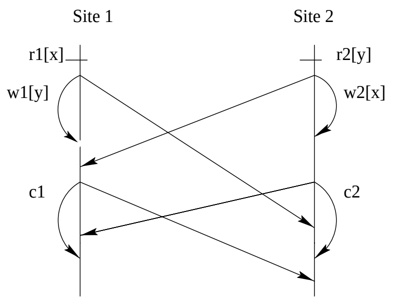

Figure 1: Incorrect execution with the early release of read locks.

In the above protocol, read locks are released at the termination (commit/abort) of a transaction. Since read operations are usually more frequent in comparison to write operations, it is advantageous to release read locks as early as possible. In the strict two-phase locking protocol, which is widely used in commercial database management systems, read locks are released when a transaction initiates commitment. In order to incorporate early release of read locks, we modify the above protocol so that read locks can be released at the time when the commit request is broadcast from the initiating site of a transaction. However, simply releasing read locks early may result in incorrect executions since read locks are maintained until termination to detect read-write conflicts with concurrent write operations. In Figure 1, we illustrate an example of such an inconsistency. At site 1, transaction T_1 executes $r_1[x]$ followed by $w_1[y]$, which is broadcast and then T_1 decides to commit. A commit message, c_1, is broadcast after releasing T_1's read lock on x. Likewise, at site 2, transaction T_2 executes $r_2[y]$ followed by $w_2[x]$, which is broadcast and then T_2 broadcasts c_2 after releasing its read lock on y. After T_1 releases its read lock on x, $w_2[x]$ is executed, and a positive acknowledgment is sent to T_2 in response to T_2's commit request, which results in T_2 committing successfully at all sites. Symmetrically, T_1 will also receive positive acknowledgments and commit. However, this execution is nonserializable.

To capture this inconsistency, we piggyback on the commit message a *read set*, which contains all objects read by a transaction. This read set is compared at all receiving sites with all write operations of locally initiated transactions thus detecting any read-write conflicts. We resolve the conflict by denying the commit request. If in the example illustrated in Figure 1, the read set is piggybacked on the commit messages, then both T_1 and T_2 are aborted (e.g., in the case of T_1, the read set is $\{r_1[x]\}$, which intersects with $w_2[x]$). The modified commit rule can now be stated as follows:

> Commit Rule.[2] When a site I decides to commit an updating transaction T_i, it reliably broadcasts a commit request c_i to all the database sites including itself. *All read locks held by T_i are released and a read set containing the identity of objects read by T_i is included in the commit request c_i.* On delivery of c_i, a site S checks if T_i has any pending write operations (note that write operations on remote sites may be blocked due to conflicting read or write operations of other transactions) or *if the read set of T_i intersects with the write sets of local transactions.* If this is the case, S broadcasts a negative acknowledgment to all the database sites. Otherwise, a positive acknowledgment is broadcast to all the database sites. A transaction is aborted if there are any negative acknowledgments to the commit request. On the other hand, if all the database sites acknowledge positively, T_i is committed and all write locks held by T_i are released.

[2]The emphasized text indicates the changes that are made to the commit rule of the first protocol.

The commit decision of read-only transactions is not broadcast and because read operations are not aborted due to read-write conflicts, read-only transactions are never aborted.

The early release of read locks allows the execution of writes conflicting with read operations during the commit phase of the reading transaction. In this sense, the above rule eliminates blocking due to read locks and therefore may avoid unnecessary aborts of remote transactions on account of pending writes. We illustrate this using the example shown in Figure 1, with the only change that the second transaction does not execute any local read operations. At site 1, transaction T_1 executes $r_1[x]$ followed by $w_1[y]$, which is broadcast, and then T_1 broadcasts c_1 (piggybacked with $\{r_1[x]\}$) after releasing its read lock on x. At site 2, transaction T_2 executes $w_2[x]$, which is broadcast and then T_2 broadcasts c_2 with an empty read set. When T_1 releases its read lock on x $w_2[x]$ obtains the write lock on x and is executed. Furthermore, when c_2 arrives, site 1 responds positively, and hence T_2 commits and releases its lock on x. After T_2 commits on site 2, $w_1[y]$ acquires its write lock and is executed, and finally T_1 is committed too (we assume that the acknowledgments for c_2 arrive before c_1 is delivered at site 2). Note that if read locks were not released early, T_2 would have been aborted, since $w_2[x]$ would be pending.

4 Causal Broadcast-Based Protocol

In this section we substitute reliable broadcast by causal broadcast and examine the benefits resulting from the use of this more powerful broadcast primitive.

We start by using the protocol in the previous section with the only difference being that messages are exchanged using a causal broadcast primitive instead of a reliable broadcast primitive. Since the correctness argument only depends on the reliability of write and commit operations being eventually delivered at all sites, the modified protocol remains correct. The use of the different broadcast primitives offers some interesting tradeoffs in terms of the sets of executed transactions and the potential interleavings between operations. On one hand, the causal broadcast imposes more ordering restrictions on message delivery than the reliable broadcast. Hence for a set of transactions with no conflicting operations, the reliable broadcast will allow more interleavings between the operations at the different sites. In this case, all such non-conflicting transactions will commit using either protocol. On the other hand, for a pair of transactions executing conflicting operations with a causal dependency, the causal broadcast-based protocol captures the conflict as a causal dependency, and enforces it. Thus, the commitment of both transactions is ensured. The reliable broadcast-based protocol may violate this causal dependency, resulting in the unnecessary abort of transactions. We now elaborate on this and show that in an execution if a transaction under reliable broadcast commits, then it must also commit under causal broadcast but not vice-versa. The difference between using the two communication primitives is reflected in our analysis only in the case of causally related conflicting transactions.

In the case of conflicting operations that are causally related, a causal ordering induced on the delivery of messages may allow the successful execution of transactions that would otherwise be aborted. We illustrate in Figure 2, a case in which the execution of a transaction is aborted under the protocol based on reliable broadcast, but it is completed if the protocol uses causal broadcast. We first consider the execution under reliable broadcast. A transaction T_1 initiated at site 1 sends a commit request after broadcasting a write on x. The commit acknowledgment of site 2 arrives at all sites. However, the commit acknowledgment for T_1 from site 3 to site 1 is delayed. In the interim, site 2 initiates a transaction T_2 that writes x and initiates its commitment. In this case since

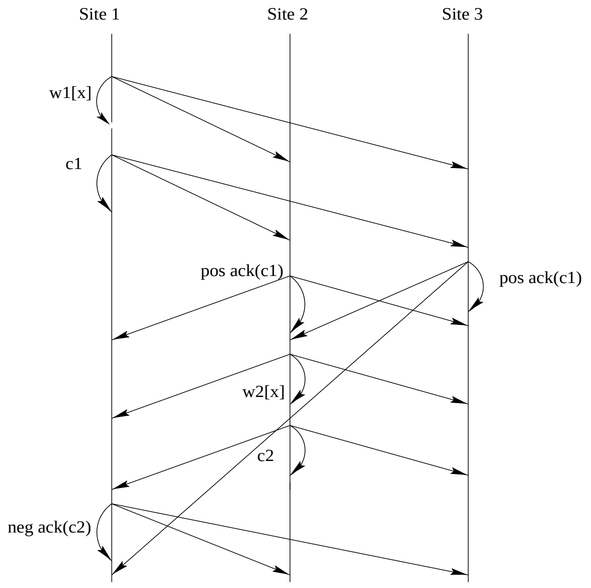

Figure 2: An example of a transaction abort under reliable broadcast, but not under causal broadcast.

site 1 is waiting for an acknowledgment message from site 3, and because it cannot commit T_1, it delays the transaction initiated at site 2. Therefore when site 2 requests to commit, site 1 finds that T_2 is pending, and hence broadcasts a negative acknowledgment as a decision to abort T_2. This scenario would not occur if instead of using reliable broadcast, sites communicate through causal broadcast. The commit acknowledgment of site 3 causally precedes the write operation initiated at site 2. Consequently on site 1, T_2 would not be pending when its commit request arrives.

We now modify the above protocol further to take advantages of causality. In particular we eliminate the need for positive acknowledgment during commitment by using the causality information as follows. When a site receives causally dependent messages on the commitment of a transaction from all the sites in the network, the receiving site can commit the transaction. On the other hand, if the site receives a negative acknowledgment the transaction is aborted. Note that in order to implement this protocol, the communication layer must expose the mechanism used for determining causal relationships among messages, e.g., the vector clocks associated with the messages [5]. Furthermore, if this information is available to the application layer, it can be also used for early detection of conflict among transactions. In particular, we can detect that two conflicting operations are concurrent and hence will be aborted. We now describe the protocol in more detail.

Causal Broadcast-Based Protocol with Implicit Positive Acknowledgments

1. Read Rule. A read operation $r_i[x]$ is executed locally by acquiring a read lock at the initiating site of transaction T_i.

2. Write Rule. A write operation $w_i[x]$ is executed by causally broadcasting it to all database sites. On delivery, a site S first checks if it can acquire a write lock on x. *If x is locked by another transaction T_j the following cases need to be considered:*

   (a) x is read locked at S. This means that T_j is initiated from S and is still in progress (because if T_j had initiated its commitment it would have released its lock at S and included x in its read set in the commit message as described earlier). In this case, all operations of T_i are delayed until granted the write lock on x.

   (b) x is write locked at S. In this case, S checks if $w_j[x]$ and $w_i[x]$ are concurrent, e.g., the vector clocks associated with $w_i[x]$ and $w_j[x]$ are incompatible. If this is not the case then T_i is causally dependent on T_j and hence the write request and all other operations of T_i are delayed at S until the lock is released by T_j. On the other hand, if $w_i[x]$ and $w_j[x]$ are concurrent, S rejects $w_i[x]$ and marks T_i for abortion.

3. Commit Rule. When a site I decides to commit a transaction T_i, it broadcasts a commit request c_i and it releases all the read locks held by T_i and c_i includes the read set of T_i. On delivery of c_i, a site checks if T_i has been marked for abortion, if it has any pending write operations, or if the read set of T_i has a non-empty intersection with the write sets of local transactions. In either case, T_i is aborted locally and a negative acknowledgment to c_i is broadcasted. *A site commits T_i only after receiving from all the sites messages that are causally dependent on c_i without receiving any negative acknowledgment for c_i.*

   Read-only transactions do not broadcast their commit decisions, and are not aborted in this protocol.

A detailed proof of correctness requires a case analysis of the different conflict cases between a pair of transactions [15].

The causal broadcast protocol with implicit positive acknowledgment presented above is most appropriate for situations where all sites broadcast messages fairly frequently; otherwise the wait for "implicit" acknowledgments can become a drawback resulting in substantial delays for transaction commitment. If instead, the protocol with explicit acknowledgment is used to circumvent this problem then the number of messages involved in the commit phase of each transaction become proportional to the square of the number of sites. We now develop a protocol that localizes commit decisions and therefore eliminates the need for acknowledgments. As in the preceding protocols, read operations are all executed locally in order to maintain the low number of messages broadcasted among sites. If we are to eliminate the need for acknowledgments, then we need a consistent way to order conflicting transactions at all sites. This order can be based on a total order of the delivered commit operations, together with consistent decisions regarding execution after their delivery.

One way to ensure a total order of commit requests is to use atomic broadcast for totally ordering commit operations. In this case, the system must support both atomic as well as causal broadcast primitives. For example, ISIS provides both primitives for application design [5]. While commit requests are delivered in total order by atomic broadcasts, the operations that are sent through causal broadcast may be con-

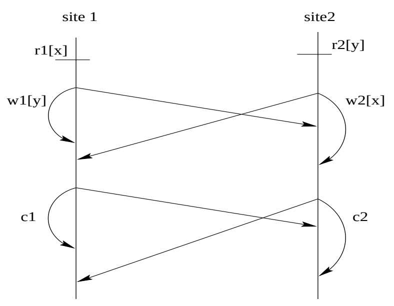

Figure 3: Incorrect execution if acknowledgments are replaced by an atomic broadcast of commit in the Causal Broadcast-Based Protocol.

current. As in the preceding protocol, concurrency of operations can be detected if the underlying communication system enables the application layer to detect concurrency of messages.

Unfortunately, a straightforward modification of the previous protocol to include the atomic delivery of commit requests does not ensure correctness. Consider the example presented in Figure 3. Site 1 would not notice the conflict between the set of reads of T_2 with its own updates because by the time it receives operation c_2, T_1 has already been committed. It would therefore commit both transactions. On the other hand, on site 2 the set of reads of T_1 intersects with the local writes, and as a consequence T_1 would be aborted. The two sites end up in inconsistent states with respect to each other by committing and aborting different transactions. In this case, the problem arises from the fact that read-write conflicts are localized. However, in order for all sites to make the same decisions regarding the commit/abort of transactions, they all need to be aware of all conflicts, both read-write and write-write. In order for all sites to receive information about read operations, as in the preceding protocol, we require that a read set be attached to the outgoing commit request. However, this is not enough, all sites also have to be able to distinguish which read operations are obsolete, that is whether a write operation follows and conflicts with the read operation on the initiating site of the reading transaction. As a possible solution, with each item we associate a version number that is incremented with every update. Version numbers also store the transaction id of the transactions that last updated the respective data items, and are included in the read set attached to the commit messages. Going back to the example illustrated in Figure 3, both sites receive first c_1 together with T_1's set of reads. They both determine that $r_1[x]$ is obsolete, but because c_1 is the first operation delivered out of the requests for committing the two conflicting transactions, they both decide on the order $T_1 \rightarrow T_2$ of transactions, and commit T_1. On the receipt of c_2, both sites again notice that $r2[y]$ is obsolete, and because this would imply a transaction order that conflicts with $T_1 \rightarrow T_2$, they consistently decide to abort T_2. Another case in which sites may reflect different conflicts is after the delivery of conflicting concurrent writes. As in the previous protocol, this case can be easily detected with the use of vector clocks, and inconsistencies can be avoided. Only the transaction involved in the conflict that has its commit request delivered first is executed, while the other conflicting transactions are aborted. More details of the algorithm are provided in [15].

The use of atomic broadcast for commit operations eliminates the need for negative acknowledgments which was necessary to inform transactions when they were involved in conflicting concurrent write operations and in read-write conflicts. Such operations are resolved by relying on the total order enforced by the atomic broadcast. However, the advantage obtained from the elimination of acknowledgments is achieved at the price of aborting at times read-only transactions. Read-only transactions do not broadcast any operations, therefore only the initiator of the reading transaction is aware of the possible conflicts during the transaction's execution. Deadlocks can occur, but they are resolved in a consistent manner. If a reading transaction is involved in a deadlock on a site S, then the only way to resolve the deadlock without aborting the read-only transaction is by aborting the conflicting writing transaction. Since this protocol does not use negative acknowledgments, other sites are not made aware of this conflict and deadlock could occur. Therefore, in order to maintain a consistent execution at all sites, our protocol resolves deadlocks that involve read-only transactions by aborting the reading transaction. Commit/abort decisions are made consistently at all sites, because in the case of a write operation blocked by a read the commit of the reading transaction could not have been broadcast. Otherwise the read locks would have been released, and in the case of a read-only transaction, the transaction would have been committed.

5 Discussion

Our attempt in this paper is to exploit the usefulness of group communication primitives in the context of a particular application, namely, transactional-based replicated databases. We make the following observations regarding our development. First, the stronger the broadcast primitive, the "better" the resulting replica control protocol from the database application perspective. Second, the protocols presented in this paper have the desirable property that either read-only transactions are not aborted, or the need for acknowledgments is completely eliminated. We have discovered in our development that the application layer should not be completely isolated from the underlying communication subsystem. In particular, some of our protocols rely on the underlying layer providing the mechanism for detecting the causality relationships among broadcast messages. In conclusion, this work represents a first step towards the study of the usefulness of broadcast primitives in distributed replicated databases. Empirical evaluation is needed to clearly establish the trade-offs between replica control protocols based on group communication and point-to-point communication.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting Atomic Broadcast in Replicated Databases. In *Proceedings of the 1997 EURO-PAR International Conference on Parallel Processing*, pages 496–503, August 1997.

[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] O. Babaoglu, A. Bartoli, and G. Dini. Replicated file management in large-scale distributed systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms, WDAG'94*, Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, October 1994.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[5] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Press, 1994.

[6] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1996.

[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] 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.

[10] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[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] A. Schiper and A. Sandoz. Primary Partition "Virtually-synchronous Communication" harder than Consensus. In *Proceedings of the 8th International Workshop on Distributed Algorithms, WDAG'94*, Lecture Notes in Computer Science, pages 39–52. Springer-Verlag, October 1994.

[14] D. Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 133–147, June 1982.

[15] I. Stanoi, D. Agrawal, and A. El Abbadi. Using Broadcast Primitives in Replicated Databases. Technical report, Department of Computer Science, University of California at Santa Barbara, 1997.