Partial Replication in the Database State Machine*

António SOUSA
Universidade do Minho
als@di.uminho.pt

Fernando PEDONE
Hewlett-Packard Labs
pedone@hpl.hp.com

Rui OLIVEIRA
Universidade do Minho
rco@di.uminho.pt

Francisco MOURA
Universidade do Minho
fsm@di.uminho.pt

Abstract

This paper investigates the use of partial replication in the Database State Machine approach introduced earlier for fully replicated databases. It builds on the order and atomicity properties of group communication primitives to achieve strong consistency and proposes two new abstractions: Resilient Atomic Commit and Fast Atomic Broadcast.

Even with atomic broadcast, partial replication requires a termination protocol such as atomic commit to ensure transaction atomicity. With Resilient Atomic Commit our termination protocol allows the commit of a transaction despite the failure of some of the participants. Preliminary performance studies suggest that the additional cost of supporting partial replication can be mitigated through the use of Fast Atomic Broadcast.

1. Introduction

Database replication protocols based on group communication primitives have recently been the subject of a considerable body of research [2, 18, 1, 19, 11, 16, 10, 6]. The reason for this stems from the adequacy of the order and atomicity properties of group communication primitives to implement *synchronous replication* (i.e., strong consistent) strategies. Unlike database replication schemes based on traditional transactional mechanisms, group-based replication mechanisms use *atomic broadcast* primitives to broadcast transactions to all replicas of the database. The approach allows the delegation of much of the synchronization complexity to the group communication layer and can accommodate different replication strategies.

Most previous work related to group-based database replication [2, 18, 1, 19, 11, 16, 10] considers full replication strategies, i.e., the whole database is available at every replica. This paper investigates the use of partial replication in the context of the Database State Machine (DBSM) [16]. Partial replication is usually favored, or even required, by environments exhibiting strong access locality. Representative examples of such settings are geographically dispersed information systems with location-dependent database sites (*eg.* banking, public administration) and large-scale distributed information retrieval systems [13].

Our approach is to extend the Database State Machine protocol to handle partial replication while preserving its replication characteristics, namely synchronous replication strategy and the deferred update technique. Synchronous replication strategies extend the atomicity concept of transactions to all database sites, instead of applying it only locally at each database. Unlike asynchronous replication strategies, synchronous strategies ensure serializable executions. Deferred update techniques execute transactions locally at some database site, and when the commit operation is requested for a transaction, the site where the transaction executed communicates the transaction to all the other sites, reducing the communication over-

*Research partially supported by FCT, ESCADA project (POSI/33792/CHS/2000).

head.

To handle partial replication efficiently in the Database State Machine, we introduce in the paper two abstractions: Resilient Atomic Commit and Fast Atomic Broadcast. Resilient Atomic Commit extends traditional atomic commit protocols to be used with data replication. Roughly speaking, Resilient Atomic Commit requires that only a subset of the sites storing a copy of the data updated by a transaction vote for the commit of the transaction. Fast Atomic Broadcast exposes preliminary message delivery orders to the application before providing the application with a final definitive order — an idea that generalizes the broadcast protocol presented in [12].

Previous work [2, 18, 1, 19, 11, 16, 10] concentrates on *full replication* strategies. Along with the assumption of the deterministic processing of transactions at every replica, the resulting protocols, characterized as *non voting* [21], take advantage of not requiring a termination protocol such as Atomic Commit [7]. In a partial replication scenario where each replica only holds a subset of the database, even when using atomic broadcast, transaction's commit atomicity requires a termination protocol such as Atomic Commit (See Section 4.1). Otherwise replicas may not agree on transaction's outcome.

To the best of our knowledge, the work in [6] is the only one, apart from ours, to consider partial database replication with group communication protocols. The approach uses group communication primitives to *immediately* broadcast read operations to all replicas of an item, and broadcast all write operations along with the transaction's commit request. Transaction atomicity is ensured by a final atomic commit protocol. By contrast, we eliminate replica interaction during transaction processing by using only one atomic broadcast message per transaction when commit is requested. Furthermore, we investigate whether the atomic broadcast can be executed concurrently with the termination protocol in an attempt to lower execution times.

The rest of the paper is structured as follows: we start by defining in Section 2 our model of the system and the abstractions upon which our solution is based. Section 3 recalls with some detail the Database State Machine protocol. Section 4 extends the DBSM to handle partial replication. Section 5 describes a prototype of the extended DBSM and presents performance measures. Section 6 concludes the paper.

2. System Model and Definitions

In this section, we present the system model and introduce Resilient Atomic Commit and Fast Atomic Broadcast, two abstractions used throughout the paper.

2.1. Databases and Transactions

We consider a system $S = \{s_1, \dots, s_n\}$ of database sites. Sites communicate through message passing (i.e., no shared memory). The system is asynchronous in that we make no assumptions about the time it takes for a site to execute a step nor the time it takes for messages to be transmitted.

Sites may only fail by crashing (i.e., no Byzantine failures), and we do not rely on site recovery for correctness. A site that never crashes is *correct*, and a site that is not correct is *faulty*. We assume that our asynchronous model is augmented with a Failure Detector Oracle [5] so that Atomic Broadcast — defined next — can be solved.

A database $\Gamma = \{x_1, \dots, x_n\}$ is a finite set of data items. Database sites have a partial copy of the database. We assume that for each data item $x_i \in \Gamma$ there is at least one correct site that stores x_i. For each site $s \in S$, $Items(s)$ is defined as the set of data items replicated in s; the set of all database sites replicating a data item $x_i \in \Gamma$ is denoted by $Sites(x_i)$.

A transaction is a sequence of read and write operations followed by a commit or abort operation, issued by a client on behalf of the transaction. Every transaction belongs to the set T of all possible transactions. For each transaction $t \in T$, $Items(t)$ is defined as the set of data items read or written by t. $RS(t)$ denotes the set of data items read by t and $WS(t)$ the

set of data items written by t. Furthermore, we denote $RS(t).s$ and $WS(t).s$ the data items read or written, respectively, by t and stored in a particular database site s.

For the sake of simplicity, we consider a replication model where a transaction t can only be executed at a site s if $Items(s) \supseteq Items(t)$, that is, s contains all data items read or written by t. This assumption can be released by allowing sites to re-direct transaction requests to other sites. Finally, $Sites(t)$ denotes the set of sites that contain data items read or written by t.

2.2. Atomic Commit and Resilient Atomic Commit

In order to ensure consistent termination of distributed transactions, database systems usually recur to an Atomic Commit protocol [7]. When each transaction participant must reach a decision despite the failure of other participants, Non-Blocking Atomic Commit protocols (NB-AC) [3], or, as presented next, Weak Non-Blocking Atomic Commit protocols [8], are used.[1]

In the (Weak Non-Blocking) Atomic Commit problem, every participant starts by voting *yes* or *no* and can reach one of two decisions: *commit* or *abort*. A NB-AC protocol is an algorithm fulfilling the following properties:

Agreement. No two participants decide different outcomes.

Termination. Every correct participant eventually decides.

Validity. If a participant decides *commit*, then all participants have voted *yes*.

Non-Triviality. If all participants vote *yes*, and no participant is ever suspected to have failed, then every correct participant eventually decides *commit*.

[1]Throughout the paper we refer to Weak Non-Blocking Atomic Commit as simply "Atomic Commit".

In the above specification, the suspicion[2] of a single participant may lead the remaining ones to decide to abort a transaction regardless of the participants votes. If data items are replicated, this means that if at least one site storing a data item read or written by a transaction is suspected, the transaction can be aborted. This clearly goes against the motivation for replicating data items — the more replicas a data item has, the higher the chances of a suspicion, and the lower the chances that transactions that read or write this data item will be committed.

Resilient Atomic Commit solves this problem by allowing participants to decide *commit* even if some of the replicas of a data item read or written by the transaction are suspected to have failed. Resilient Atomic Commit satisfies the same agreement and termination properties of Weak Non-Blocking Atomic Commit and the following validity and non-triviality properties:

Validity: If a site decides *commit* for t, then for each $x \in Items(t)$, there is at least a site in $Sites(x)$ that voted *yes* for t.

Non-triviality: If for each $x \in Items(t)$ there is at least a site $s \in Sites(x)$ that votes *yes* for t and is not suspected, then every correct site eventually decides *commit* for t.

2.3. Atomic Broadcast and Fast Atomic Broadcast

Atomic Broadcast and Fast Atomic Broadcast are the communication abstractions used by database sites to communicate. Atomic Broadcast is defined by the primitives *broadcast*(m) and *deliver*(m), and satisfies the following properties [9]:

Validity. If a correct site broadcasts a message m, then it eventually delivers m.

Agreement. If a correct site delivers a message m, then every correct site eventually delivers m.

[2]This information is provided locally to each participant by the Failure Detector Oracle [5].

Integrity. For every message m, every site delivers m at most once, and only if m was previously broadcast.

Total Order. If two correct sites deliver two messages m and m', then they do so in the same order.

When using an atomic broadcast primitive, all sites must wait until they agree on message order before atomically delivering it. In the following, we present *Fast Atomic Broadcast*, which allows sites to deliver messages tentatively, that is, before the order has been agreed.

Fast Atomic Broadcast is defined by the primitives *broadcast(m)*, *FST-deliver(m)*, and *FNL-deliver(m)*, which satisfy the following properties:

Validity. If a correct site broadcasts a message m, then it eventually FNL-delivers m.

FST-Agreement. For any $k \geq 0$, if a correct site FST-delivers a message m k times, then every correct site also FST-delivers m k times.

FNL-Agreement. If a correct site FNL-delivers a message m, then every correct site eventually FNL-delivers m.

Integrity. For every message m, every site FST-delivers m only if m was previously broadcast; and every site FNL-delivers m only once, and only if m was previously broadcast.

Local Order. No site FST-delivers a message m after having FNL-delivered m.

Final Order. If two sites FNL-deliver two messages m and m', then they do so in the same order.

Fast Atomic Broadcast allows sites to *guess* the definitive order of messages and expose this order to the application. The application can then start treating the message concurrently with the underlying ordering mechanism used by Fast Atomic Broadcast to finally order the message. Notice that if a site FST-delivers a message and then changes its initial guess, it may FST-deliver the message again. Obviously, applications must be able to cope with messages FST-delivered in the wrong order.

Figure 1 compares the execution, as seen by the application, of Atomic Broadcast and Fast Atomic Broadcast. In Figure 1(a) messages are broadcast and delivered to the sites only when their order is determined, while in Figure 1(b), messages are FST-delivered twice before being FNL-delivered.

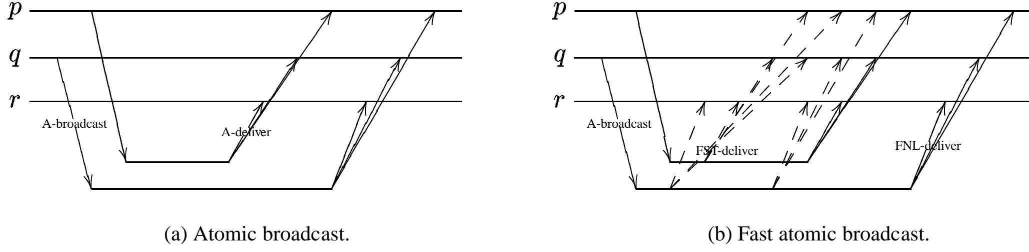
Fast Atomic Broadcast is similar to Atomic Broadcast with Optimistic Delivery, introduced in [12]. Actually, Atomic Broadcast with Optimistic Delivery is a special case of Fast Atomic Broadcast, where $k = 1$ (see the FST-Agreement property).

3. Database State Machines

The Database State Machine [16], or DBSM, assumes the full replication of the database and is based on the deferred update replication technique [3]. In this section we recall the principle of the deferred update replication and the DBSM approach.

3.1. Deferred Update Replication Principle

The deferred update replication technique is a way to reduce the need for distributed coordination among concurrent transactions during their execution. Using this technique, a transaction is locally synchronized during its execution at the database where it initiated according to some concurrency control mechanism [3] (e.g., two-phase locking). Interaction with other database sites on behalf of the transaction only occurs when the client requests the transaction commit. At this time, the transaction updates and some control structures are propagated to all database sites. Each such database site will then *certify* and, if possible, commit the transaction. The *termination protocol*, started with the commit request, has three goals: (*i*) propagate the transaction to all database sites, (*ii*) cer-



(a) Atomic broadcast.

(b) Fast atomic broadcast.

Figure 1. Atomic Broadcast vs. Fast Atomic Broadcast.

tify, and (*iii*) commit it.

3.2. Transaction Execution

From the time it starts until it finishes, a transaction passes through some well-defined states (Figure 2). The starting state is the *executing state*, a state where all operations are executed locally at the database site where the transaction starts. When the client that initiates the transaction requests its commitment, the transaction passes to the *committing state*. At this point, transaction t, is sent to all database sites. A transaction received by a database site s is in the committing state until its fate is known. The transaction then evolves to one of its final states *committed* or *aborted*.

The algorithm executed by a database site s_i when executing a transaction received from client c is briefly described as follows:

1. Initially, during the *executing* state, the transaction is locally executed at database site s_i. All operations requested by client c are executed at s_i using strict two-phase locking.

2. When client c requests transaction t's commitment, t is immediately committed if it is a read-only transaction. Otherwise, t enters the committing state and database site s_i starts the termination protocol for t: the updates performed by t, as well as its readset and writeset, are broadcast to all database sites.

3. Eventually every database site s_j delivers the message sent by s_i concerning transaction t. After delivering this message, s_j starts t's certification to ensure that t it does not *conflict* with previously committed transactions.

4. If t passes the certification test, all t's updates are applied to the database and t passes to the committed state. Transactions in the execution state at s_j holding locks on data items updated by t are aborted.

5. The database site s_i sends t's result to client c as soon as s_i establishes the final state of t.

3.3. Conflicting Transactions

In order for a database site to certify a committing transaction t, it must be able to determine which transactions conflict with t. A transaction t' *conflicts with* t if: (*i*) t and t' have conflicting operations and (*ii*) t' does not *precede* t.

Two operations conflict when they are issued by different transactions, access the same data item and at least one of them is a write operation. The precedence relation between transactions t and t' is denoted $t' \rightarrow t$ (i.e., t' precedes t) and defined as: (1) if t and t' execute at the same database site, t' precedes t if t' enters the committing state before t; or (2) if t and t' execute at different sites, for example s_i and s_j, respectively, t' precedes t if t' commits at s_i before t enters the committing state at s_i.

Furthermore, we say that two transactions are *write-conflicting* if they both perform a write operation on the same data item and one transaction does

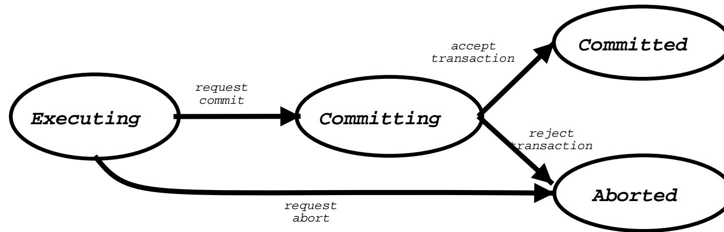


Figure 2. Transaction states

not precede the other.

3.4. DBSM Architecture

Transaction processing in the DBSM [16] is handled by the *Transaction Manager*, the *Lock Manager*, and the *Data Manager* modules presented in Figure 3. The termination protocol is handled by the *Atomic Broadcast*, and the *Certification* modules.

After receiving a transaction delivered by the Atomic Broadcast module, the certification module executes the certification test. On certifying a transaction, the data manager may be inquired about already committed transactions. If the transaction is successfully certified, its write operations are transmitted to the lock manager, and, once the write locks are granted, the updates can be performed.

To ensure that each database site reaches the same state after processing committing transactions, each certification module has to (*i*) reach the same decision when certifying transactions, and (*ii*) guarantee that write-conflicting transactions are applied to the database in the same order. The first constraint can be fulfilled by providing each certification module with the same set of transactions in the same order. To satisfy the second constraint, the certification module ensures that write-conflicting transactions grant their locks in the same order as they are delivered.

4. Handling Partial Replication

In this section we consider partial replication in the context of the DBSM. We point out that the DBSM as it is does not support partial replication and discuss ways of extending the termination protocol to handle partial replication. We start with a simple approach based on Atomic Broadcast and Atomic Commit, and then refine it to reach more sophisticated solutions based on Fast Atomic Broadcast and Resilient Atomic Commit.

4.1. DBSM and Partial Replication

The DBSM assumes that databases contain full copies of all data items. This assumption is necessary to make sure that upon certifying a transaction, all database sites reach the same decision, whether to commit or abort the transaction. As we show next, partially replicated data items may lead to inconsistencies, with some databases deciding to commit a transaction and some deciding to abort it.

For example, consider a system composed of three database sites, s_1, s_2, and s_3 — database site s_1 replicates data items a and b, database site s_2 replicates data items b and c and database site s_3 replicates data items a and c — and two clients c_1 and c_2 which submit, respectively, transactions $t_1 = \langle r[a]; w[a]; w[b]; c\rangle$ and $t_2 = \langle r[a]; w[a]; w[c]; c\rangle$.

If transactions t_1 and t_2 are executed concurrently in different databases (i.e., neither t_1 precedes t_2 nor t_2 precedes t_1) and t_2 is delivered and certified before t_1, t_2 commits at all sites while t_1 commits at s_2 (i.e., $WS(t_2).s_2 \cap RS(t_1).s_2 = \emptyset$ at s_2) and aborts at s_1 (i.e., $WS(t_2).s_1 \cap RS(t_1).s_1 = \{a\}$ at s_1) and at s_3 (i.e., $WS(t_2).s_3 \cap RS(t_1).s_3 = \{a\}$ at s_3).

At first glance, one way of solving this problem is

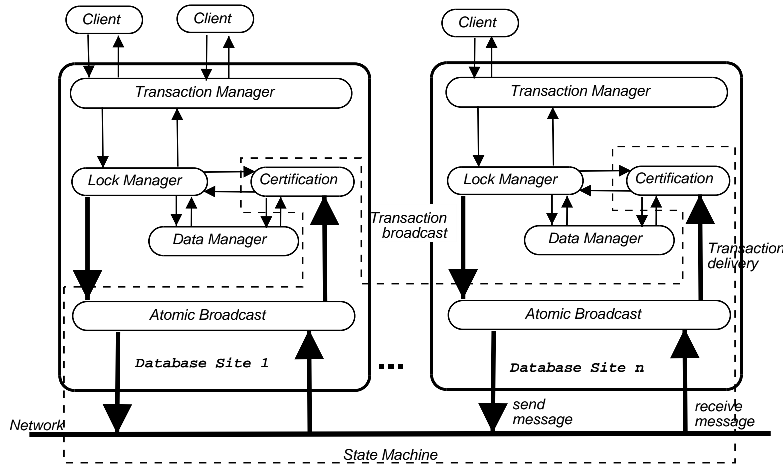


Figure 3. DBSM architecture

to have every database to store all the read and write sets of previously committed transactions so that the certification test performed by each database results in the same outcome, as it is done with the DBSM; however, such an approach would have the overhead of full replication (i.e., every database has to keep track of all data items read and written by transactions) without its benefits. Since databases do not store all data items, transactions cannot execute in any database!

4.2. DBSM with Atomic Commit

As discussed, database sites that hold a partial copy of the data items cannot decide to commit a transaction based only on the certification test — they should also consider data items stored in other database sites and decide on a common basis. This is typically done by an atomic commit protocol, and, in this case, each database should use the result of the certification test as its vote for the atomic commit protocol.

The certification of a transaction involves now *(i)* a *certification test* and *(ii)* an *atomic commit* among the database sites that store copies of the data items used by the transaction (see Figure 4). The procedure of certifying a transaction t at database site s_i is described as:

1. *Certification test.* The certification test at database site s_i involves every data item accessed by t for which s_i holds a replica. Database s_i votes *yes* if all committed transactions at s_i precede t, or if there is no committed transaction t' at s_i that conflicts with t; s_i votes *no* otherwise. The vote of site s_i on transaction t is formally described as follows.

$$vote_i(t) \equiv \left[\begin{array}{l} \forall t',\ Committed(t', s_i) : \\ \\ t' \rightarrow t \vee (WS(t').s_i \cap RS(t).s_i = \emptyset) \end{array} \right]$$

2. *Atomic commit.* The atomic commit protocol is executed by all database sites holding a replica of a data item accessed by the transaction. After applying the certification test to transaction t, every database site s_i involved in t's commit starts an atomic commit using as its vote the outcome of the certification test. If the result of the atomic commit protocol is *commit*, then t passes to the commit state at s_i, all updates issued by t for data items stored in s_i are performed, and the locks associated with t released.

Atomic Broadcast-based termination. The need to execute an atomic commit as part of the certification procedure leads to question the necessity for the

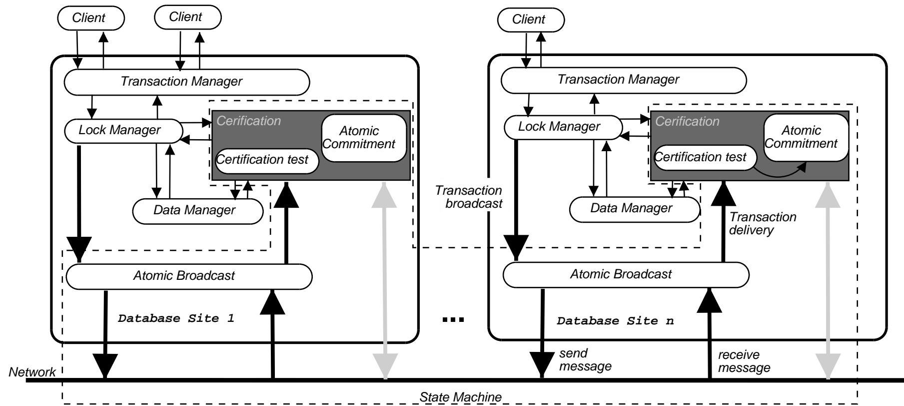


Figure 4. DBSM with Atomic Commit

Atomic Broadcast used in the beginning of the termination protocol. Instead of ordering distributed transactions before certifying them, one might simply forward the transactions to all sites without any ordering guarantees. It turns out, however, that ordering transactions before certifying them allows a more efficient certification test [15].

For example, consider again the case presented in Section 4.1, and assume that both transactions t_1 and t_2 start their termination protocols concurrently — that is, t_1 (respectively, t_2) is forwarded to the other sites before t_2 (respectively, t_1) is certified. Since t_1 and t_2 conflict, they cannot be both committed, and one of them should be aborted. But because databases do not necessarily receive and certify t_1 and t_2 in the same order, some databases may certify t_1 first and vote to commit t_1 and abort t_2, while others may certify t_2 before t_1, and vote to commit t_2 and abort t_1, a situation where both transactions end up aborted.

4.3. DBSM with Resilient Atomic Commit

The combination of atomic broadcast and atomic commit enables to support partial replication without compromising consistency. However, with such an approach, the suspicion of a single database site is enough to abort a transaction (see the non-triviality property of atomic commit), which defeats the purpose of introducing replication. In fact, such a replicated system is *less* resilient than a non-replicated one. This approach also introduces extra overhead — the execution of the atomic commit protocol.

In order to overcome the former problem, i.e., committing transactions even when some database site is suspected to have crashed, we replace atomic commit by Resilient Atomic Commit in the termination protocol. With Resilient Atomic Commit, a transaction t passes to the committed state at every site s in $Sites(t)$ if: every database site holding a replica of a data item accessed by t either votes *yes* for t or is suspected; and for each data item read or written by t, there is a site that votes *yes* for t and is never suspected.

Figure 5 depicts the execution of transaction t, which is committed using DBSM with Resilient Atomic Commit but aborted if using DBSM with Atomic Commit. In step 1, transaction t executes at database site s_1, and client c sends a commit request to the database site s_1. In step 2, t is broadcast and at the end of this step, it is delivered, certified and s_2 crashes. Sites s_1 and s_3 start the Resilient Atomic Commit protocol voting *yes* and using s_1 as coordinator, which decides *commit* at the end of step 3 (using Atomic Commit, the transaction will be aborted since s_2 is eventually suspected to have failed). In step 4, s_1 sends its decision to all database sites. In step 5, database sites s_1 and s_3 receive the decision of the Resilient Atomic Commit and s_1 sends the transaction

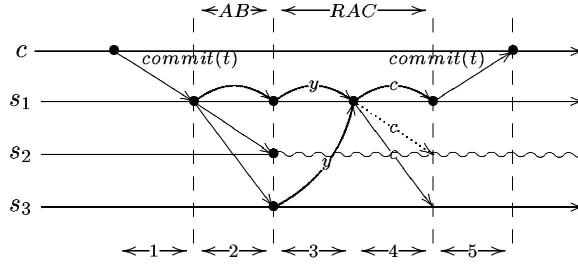


Figure 5. DBSM with Resilient Atomic Commit

result to c.

4.4. DBSM with Fast Atomic Broadcast

The addition of an atomic commitment step in the termination protocol introduces an unavoidable overhead. To alleviate this problem we replace the Atomic Broadcast protocol with a Fast Atomic Broadcast protocol. The idea is simple and consists in starting the certification process earlier, as soon as the transaction is delivered with a tentative order. Whenever this tentative order matches the final delivery, this allows us to overlap the final delivery (FNL-deliver) of the transaction with the certification test and the atomic commit protocol.

In more detail, the protocol runs as follows. When a transaction t is broadcast, it is FST-delivered to all replicas in $Sites(t)$ with a tentative order. This order is expected to be the network's spontaneous order, i.e. not yielded by the ordering algorithm, and thus allowing a fast delivery. As soon as t is FST-delivered at a site s, s starts t's certification and afterwards a resilient atomic commit for t. Upon the FNL-delivery of t, if the final and tentative orders match then the outcome of the antecipated atomic commit is used to decide the final state of t. Should the orders of the two deliveries mismatch, both the certification and the atomic commit started for t are discarded and the process repeated for the final order.

In our current prototype of the system (Section 5), any transaction t' that might be FST-delivered between the FST-deliver and the FNL-deliver of some transaction t is discarded. While this might seem a clear loss of opportunities by the protocol, doing it differently involves further research as discussed in Section 6.

5. Prototype and Results

In this section we describe a prototype of the DBSM extended to support partial replication. Performance results show how the use of a Fast Atomic Broadcast primitive mitigates the overhead introduced by the additional atomic commit protocol.

5.1. Implementation

The prototype strictly follows the architecture depicted Figure 4: a transaction processing module consisting of a transaction manager, a lock manager, a data manager, and a certification module. The atomic broadcast and atomic commit modules have been built as separate modules to independently allow several implementations, i.e., different combinations of atomic broadcast and atomic commit protocols can be used by the prototype.

The prototype has been implemented in JAVA, using the GROUPZ group communication toolkit [17].

Concurrency control and conflict detection is performed by a lock manager accessed by transactions either running locally or being certified. Database access is done using a data manager which has been implemented using JDBC [20] to access a PostgreSQL [14] database. The concurrency control mechanisms of PostgreSQL are not used as, in our model, remote transactions have priority over local transactions and PostgreSQL concurrency control does not

distinguishes between them.

The Atomic Broadcast and Fast Atomic Broadcast protocols have similar implementations: They use GROUPZ's reliable broadcast primitive[3] and a sequencer database site determined by GROUPZ's Group Membership service. When a transaction's commit is requested, the transaction is broadcast. In the Fast Atomic Broadcast this message is FST-delivered at every (correct) site. The distinguished site acting as sequencer assigns the message's order and reliably broadcasts it. When delivered (or FNL-delivered in the Fast Atomic Broadcast protocol) the message provides the ordered transaction.

The Resilient Atomic Commit is a simple n to n, single step, decentralized protocol. When starting the protocol every participant broadcasts its vote, and starts gathering votes from the other participants until it can reach a decision.

5.2. Experiments

For our experiments we used a database of 2000 data items considered as hot-spots of a larger database — we chose a relatively small database to introduce a reasonable amount of data contention in the database. The transactions submitted by clients contain between 5 and 10 operations. Update transactions, with 50% of write operations, represent 95% of all submitted transactions. We used a 100 Mb/s local-area network consisting of ten 333MHz Intel-based processor machines with 128MB of RAM running the Linux operating system.

The tests aim to compare the performance of the system using either Atomic Broadcast and Fast Atomic Broadcast followed by a Resilient Atomic Commit protocol. The graphs in Figure 6 present the histograms of transaction execution times using Atomic Broadcast and Resilient Atomic Commit (Figure 6(a)) and Fast Atomic Broadcast and Resilient Atomic Commit (Figure 6(b)). Figure 6(a) presents curves for the atomic delivery of transactions, the end of the certification execution, the end of the atomic commit protocol and the end of the transaction execution. Figure 6(b) also includes the fast delivery of transactions.

Both tests were run under a system workload of 5 tps, allowing a stable flow of transactions without queuing. Certification consisted in the management of a lock table resident on disk and accounts for an average of 20 ms of each transaction processing time. Messages did not suffered from reordering.

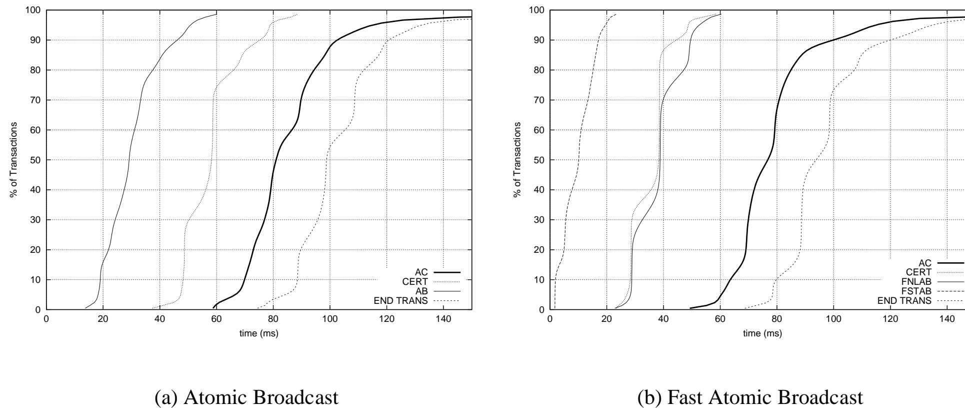
A comparison of the graphs of Figure 6 reveals that the protocol with Fast Atomic Broadcast consistently outperforms the Atomic Broadcast configuration. Indeed, it can be observed that the Fast Atomic Broadcast configuration is on average 10 ms faster. Roughly, this corresponds to a 10% gain since it can be seen that in 90% of the transactions the termination protocol finishes in less than 100 ms (Figure 6(b)).

These results are encouraging and justify the use of a Fast Atomic Broadcast primitive. However, it is worth noting that the protocol is actually very sensitive to message processing overheads and to the nature of the certification step. It can be seen in Figure 6(b) that the FNL-delivery of the transaction happens at a later time than the delivery in Figure 6(a). This is the delay introduced by the processing overhead of the fast delivery at the sequencer site. As long as the certification step and message delivery can be executed concurrently the delayed FNL-delivery does not constitute a problem.

6. Conclusions

This paper investigates the use of partial replication in the context of the Database State Machine, introduced in [16] for fully replicated databases. In order to handle partial replication efficiently, we have introduced in the paper two abstractions: Resilient Atomic Commit, an atomic commit protocol tailor-made for replicated databases, and Fast Atomic Broadcast, a communication primitive that allows applications to be

[3]This primitive is actually a View Synchronous Multicast primitive [4] ensuring *view atomicity* of the messages.



(a) Atomic Broadcast

(b) Fast Atomic Broadcast

Figure 6. Execution times for the two configurations of the termination protocol

exposed to tentative delivery orders before the final order is known.

Preliminary performance studies of our protocol using PostgreSQL [14] have shown that the introduced techniques are very promising. We intend to continue with experimental work to better understand the strengths and weaknesses of our the approach. In particular, we currently pursue two directions: one is to make the protocol more agressive regarding the fast deliveries of transactions, the other is the study of the protocol's behavior in heterogeneous large-scale networks.

As described in Section 4.4, the protocol only considers one fast delivery at a time. When treating the FST-delivery of a transaction, say t, instead of discarding a subsequent FST-delivery of a transaction t' (which may happen before a FNL-delivery), the protocol can possibly be improved in two ways. Either, consider that t and t' are both equally good candidates for the FNL-delivery and so start the certification of both transactions concurrently, or consider that t' will be FNL-delivered after the FNL-delivery of t in which case the protocol should be able to "pipeline" the certification of t' assuming the the tentative certification of t. Which method to choose is the subject of ongoing research. However, the important issue to note is that whatever is the most appropriate depends on a number of factors such as the accuracy of the network's spontaneous ordering of messages, the delay between FST and FNL-deliveries, certification costs, processing power, etc. Considering a heterogeneous large-scale network, instead of the homogenoeus local network of the experiments of Section 5, definitely introduces substantial variations on these factors.

References

[1] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), 1997.

[2] Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication. Technical Report CS94-20, The Hebrew University of Jerusalem, November 1994.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in Database Systems*. Addison-Wesley, 1987.

[4] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1), February 1987.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), Mar. 1996.

[6] U. Fritzke and P. Ingels. Systéme transactionnel pour donnés partiellement dupliqués, fondé sur la commu-

nication de groupes. Technical Report 1322, INRISA, Rennes, France, April 2000.

[7] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.

[8] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-9)*, LNCS 972, pages 87–100, Le Mont-St-Michel, France, Sept. 1995. Springer-Verlag.

[9] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[10] J. Holliday, D. Agrawal, and A. E. Abbadi. The performance of database replication with group multicast. In *Proceedings of IEEE International Symposium on Fault Tolerant Computing (FTCS29)*, pages 158–165, 1999.

[11] B. Kemme and G. Alonso. A suite of database replication protocols based on communication primitives. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

[12] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 424–431. IEEE, May 1999.

[13] Z. Lu and K. McKinley. Partial collection replication versus caching for information retrieval systems. In *The ACM International Conference on Research and Development in Information Retrieval*, Athens, Greece, July 2000.

[14] B. Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, 2000.

[15] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, Southampton, England, Sept. 1998.

[16] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. Technical Report SSC/1999/008, École Polytechnique Fédérale de Lausanne, Switzerland, March 1999.

[17] J. Pereira and R. Oliveira. Object-oriented open implementation of reliable communication protocols. In *OOPSLA'97 Workshop on*, Atlanta, USA, October 1997.

[18] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39:84–87, April 1996.

[19] I. Stanoi, D. Agrawal, and A. E. Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* 18th *IEEE International Conference on Distributed Computing Systems ICDCS'98*, pages 148–155, Amsterdam, The Netherlands, May 1998. IEEE.

[20] S. White and M. Hapner. *JDBC 2.1 API*. Sun Microsystems, December 1999.

[21] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, pages 206–215, Nürnberg, Germany, Oct. 2000. IEEE Computer Society.