Using Optimistic Atomic Broadcast in Transaction Processing Systems*

Bettina Kemme⋆ Fernando Pedone†
Gustavo Alonso⋄ André Schiper‡ Matthias Wiesmann‡

⋆School of Computer Science
McGill University
Montreal, Canada, H3A 2A7
E-mail: kemme@cs.mcgill.ca

†Software Technology Laboratory
Hewlett-Packard Laboratories, Palo Alto, CA 94304
E-mail: Fernando_Pedone@hp.com

⋄Information and Communication Systems
Swiss Federal Institute of Technology (ETH), CH-8092 Zürich
E-mail: alonso@inf.ethz.ch

‡Operating Systems Laboratory
Swiss Federal Institute of Technology (EPFL), CH-1015 Lausanne
E-mail: dragon@lsemail.epfl.ch

Index terms: Replicated Databases, Optimistic Processing, Atomic Broadcast, Transaction Processing

Abstract

Atomic broadcast primitives are often proposed as a mechanism to allow fault-tolerant cooperation between sites in a distributed system. Unfortunately, the delay incurred before a message can be delivered makes it difficult to implement high performance, scalable applications on top of atomic broadcast primitives. Recently, a new approach has been proposed for atomic broadcast which, based on optimistic assumptions about the communication system, reduces the average delay for message delivery to the application. In this paper, we develop this idea further and show how applications can take even more advantage of the optimistic assumption by overlapping the coordination phase of the atomic broadcast algorithm with the processing of delivered messages. In particular, we present a replicated database architecture that employs the new atomic broadcast primitive in such a way that communication and transaction processing are fully overlapped, providing high performance without relaxing transaction correctness.

*A preliminary version of this paper appeared in [16]. In this paper we provide a more comprehensive protocol and study the performance through simulation.

1 Introduction and Motivation

Group communication primitives are often proposed as a mechanism to increase fault tolerance in distributed systems. These primitives use different ordering semantics to provide a very flexible framework in which to develop distributed systems. One example of the available semantics is the *Atomic Broadcast* primitive [6, 4] which guarantees that all sites deliver all messages in the same order. Unfortunately, it is also widely recognized that group communication systems suffer from scalability problems [5, 8]. While performance characteristics depend on the implementation strategy, the fundamental bottleneck is the need to do some coordination between sites before messages can be delivered. This results in a considerable delay since messages cannot be delivered until the coordination step has been completed. Such delay makes it very difficult to implement high performance, scalable applications on top of group communication primitives.

Recently, a new approach has been proposed for atomic broadcast which, based on optimistic assumptions about the communication system, reduces the average delay for message delivery to the application [22]. The protocol takes advantage of the fact that in a LAN, messages normally arrive at the different sites exactly in the same order. Roughly speaking, this protocol considers the order messages arrive at each site as a first optimistic guess, and only if a mismatch of messages is detected, further coordination rounds between the sites are executed to agree on a total order. The idea has significant potential as it offers a feasible solution to the performance problems of group communication.

In this paper we develop this idea further, and show how applications can take full advantage of the optimistic assumption by overlapping the coordination phase of the atomic broadcast algorithm with the processing of delivered messages. In particular, we present a replicated database architecture that employs the new atomic broadcast primitive in such a way that communication and transaction processing are fully overlapped providing high performance without *relaxing* transaction correctness (i.e., serializability). Our general database framework is based on broadcasting updates to all replicas, and using the total order provided by the atomic broadcast to serialize the updates at all sites in the same way [1, 14, 20, 21]. The basic idea is that the communication system delivers messages twice. First, a message is preliminary delivered to the database system as soon as the message is received from the network. The transaction manager uses this tentative total order to determine a scheduling order for the transaction and starts executing the transaction. While execution takes place without waiting to see if the tentative order was correct, the commitment of a transaction is postponed until the order is confirmed. When the communication system has determined the definitive total order, it delivers a confirmation for the message. If tentative and definitive orders are the same, the transaction is committed, otherwise different cases have to be considered. If the wrongly ordered transactions do not conflict, i.e., they do not access the same objects, their execution order is irrelevant for the database system and the mismatch of tentative and definitive order has no negative effect. In the case of conflics, however, wrongly ordered transactions have to be aborted and rescheduled in order to guarantee that the serialization order obeys the definitive total order. This means that if tentative and definitive order are the same for most messages or if the probability of conflict between concurrent transactions is small, then we are able to hide some of the communication overhead behind the cost of executing a transaction.

The results reported in this paper make several important contributions. First, our solution avoids most of the overhead of group communication by overlapping the processing of messages (execution of transactions) with the algorithm used to totally order them. In environments where the optimistic assumption holds (namely local area networks) or for workloads with small conflict rates, this may be a first step towards building high performance distributed systems based on group communication primitives. Second, the transaction processing strategy follows accepted practice in database management systems in that we use the same correctness criteria (i.e., serializability) and mechanisms as existing database management systems. Third, we solve the problem of the mismatch between the total order used in group communication and the data flow ordering typical of transaction processing, thereby not losing concurrency in the execution of transactions. Finally, our approach compares favorably with existing commercial solutions for database replication in that it maintains global consistency and has the potential to offer comparable performance.

The paper is structured as follows. In Section 2 we describe the system model and introduce some definitions. In Section 3 we present the atomic broadcast primitive used in our database algorithms, and discuss degrees of optimism for atomic broadcast protocols. The basic ideas of optimistic transaction processing are described in Section 4. In Section 5, we enhance the solution to work with arbirtrary transactions. Queries are discussed in Section 6. Section 7 provides a simulation study of our approach and Section 8 concludes the paper.

2 Model and Definitions

Our formal definition of a replicated database combines the traditional definitions of distributed asynchronous systems and database systems. A replicated database consists of a group of sites $N = \{N_1, N_2, \ldots, N_n\}$ which communicate with each other by means of an atomic broadcast primitive. Sites can only fail by crashing (i.e., we exclude Byzantine failures), and always recover after a crash.

We assume a fully replicated system, i.e., each site N_i contains a copy of the entire database. The data can be accessed by executing transactions. Updates are coordinated on the replicas by two different modules: the communication module using *Atomic Broadcast with Optimistic Delivery* and the transaction management module providing *Optimistic Transaction Processing*.

2.1 Atomic Broadcast with Optimistic Delivery

Communication between sites is based on atomic broadcast. Each site broadcasts a message to all other sites. Atomic broadcast provides an *ordering* of all messages in the system, i.e., all sites receive all messages in the same order. Furthermore, *reliability* is provided in the sense that all sites decide on the same set of messages to deliver. Sites that have crashed will deliver the messages after recovering from the failure.

In this paper we consider an atomic broadcast protocol with optimistic delivery. Briefly, this protocol is based on optimistic assumptions about the network in order to deliver messages fast: if most of the time broadcast messages arrive at their destinations in the same order (a property called *spontaneous total order*) optimizations can be made that allow for protocols with better performance than traditional atomic broadcast protocols.

The atomic broadcast with optimistic delivery used in this work is formally defined by the three primitives shown below. (Further details and a complete specification of the atomic broadcast with optimistic delivery is given in Section 3.)

- `TO-broadcast(`m`)` broadcasts the message m to all sites in the system.
- `Opt-deliver(`m`)` delivers a message m optimistically to the application. `Opt-deliver` does not guarantee total order. We consider the order perceived by the application by receiving the sequence of `Opt-delivered` messages as a *tentative order*.
- `TO-deliver(`m`)` delivers m definitively to the application. The order perceived by the application by receiving the sequence of `TO-delivered` messages is called the *definitive order*.

In practice, `TO-deliver(`m`)` will not deliver the entire body of the message (which has already been done by `OPT-deliver`(m)), but rather deliver only a confirmation message that contains the identifier of m.

2.2 Transaction Model

Typically, there are three ways to interact with a relational database. One is to use SQL interactively through a console. A second one is to use embedded SQL, that is, to use SQL as part of programs written in other programming languages such as C. Program execution and flow control takes place outside the database system and only the specific database operations are transfered to the database system. A third possibility is to use stored procedures. A stored procedure allows to encapsulate complex interactions with the database into a single procedure which is executed within the database context. It can be invoked using standard remote procedure call (RPC) mechanisms. While discussing the nature and advantages of stored procedures is beyond the scope of this paper, it must be pointed out that it is an approach that greatly facilitates interaction with databases as it allows to ignore the database schema and the query language entirely. Stored procedures are written by experts and then can be easily used by programmers which do not need to know anything about the underlying database system. Since the entire code, both data manipulation and the flow control of the program, are executed within the scope of the database system, this approach leads to better performance and simplified access.[1]

We assume the traditional transaction model [3]. A transaction (stored procedure) is a sequence of read $r_i(X)$ and write $w_i(X)$ operations on objects X and executes atomically, i.e., it either commits or aborts all its results. For the moment being, we will only consider update transactions. Queries are introduced in Section 6. In our replication model, update transactions accessing an object X perform both their read and write operations on an object X on all copies of X in the system. Hence an operation $o_i(X), o \in \{r, w\}$, is translated to physical operations $o_i(X_1), \ldots, o_i(X_n)$. It is possible for two or more transactions to access the database concurrently. We say that two operations conflict if they are from different transactions, access the same copy and at least one of them is a write. A local history $H_N = (\Sigma_N, <_N)$ of a node N describes all physical operations of a set of transactions Σ being executed on the copies of N. Furthermore, it describes a partial order, $<_N$, which orders all operations within a transaction (as they are executed by the stored procedure) and additionally all conflicting operations. We only look at the committed projection of a history, which is the history after removing all active or aborted transactions. A history H is serial if it totally orders all transactions. A correct execution will be determined in terms of conflict equivalence to a serial history. Two histories, H_1 and H_2, are conflict equivalent if they are over the same set of operations and they order conflicting operations in the same way. A history is serializable if it is conflict-equivalent to a serial history. In particular, a history H is conflict-serializable if its serialization graph $SG(H)$

[1] In fact, many commercial databases base their replication solutions on the use of stored procedures [24, 9]

is acyclic (transactions are nodes in the graph, and there is an edge between T_i and T_j if T_i and T_j have conflicting operations $o_i(X)$ and $o_j(X)$, and $o_i(X)$ is executed before $o_j(X)$ in the history).

Since we use a replicated database system, *1-copy-serializability* will be the correctness criterion: despite the existence of multiple copies, an object appears as one logical copy (also called *1-copy-equivalence*) and the execution of concurrent transactions is equivalent to a serial execution over the logical copy (*serializability*).

The serializable execution of concurrent transactions is achieved through concurrency control. The concurrency control mechanisms allow non-conflicting transactions to execute in parallel while conflicting ones have to be serialized. A concurrency control protocol provides serializability when all executions it allows are serializable. Since in our model all operations of all transactions are executed on all sites, the combined concurrency control and replica control protocols provide 1-copy-serializability if the following holds: For any execution resulting in local histories $H_1, H_2, \ldots, H_n$ at sites $N_1, \ldots, N_n$, the graph $\bigcup_i SG(H_i)$ is acyclic.

2.3 Concurrency Control

Concurrency control is done via standard locking mechanisms. It is performed independently at each site. Before a transaction accesses an object, it has to acquire the corresponding read/write lock on the object. There may not be two conflicting locks granted on an object. We maintain a lock queue for each object X where the lock entries are added in FIFO order. If the first lock in the queue is a write lock it is the only granted lock. Otherwise all read locks before the first write lock are granted. Whenever a lock is released, the next lock(s) in the queue are granted. Our locking protocol requests all locks at the beginning of the transaction and releases them at the end of the transaction. Later on, we will see that this is necessary because we have to add the lock entries of a transaction in an atomic step. This atomicity can be implemented, e.g., by acquiring a latch on the lock table during the lock acquisition phase.

2.4 Execution Model

Figure 1 depicts the coordination of the communication manager and the transaction manager to execute update transactions. When a user sends the request for an update transactions to N sites, N `TO-broadcasts` the request to all sites so that the corresponding stored procedure is executed at all sites. A first module, the *Tentative Atomic Broadcast* module, receives the messages, and immediately `Opt-delivers` them to the transaction manager. In the transaction manager part of the system, the *Serialization* module takes the messages, analyzes the

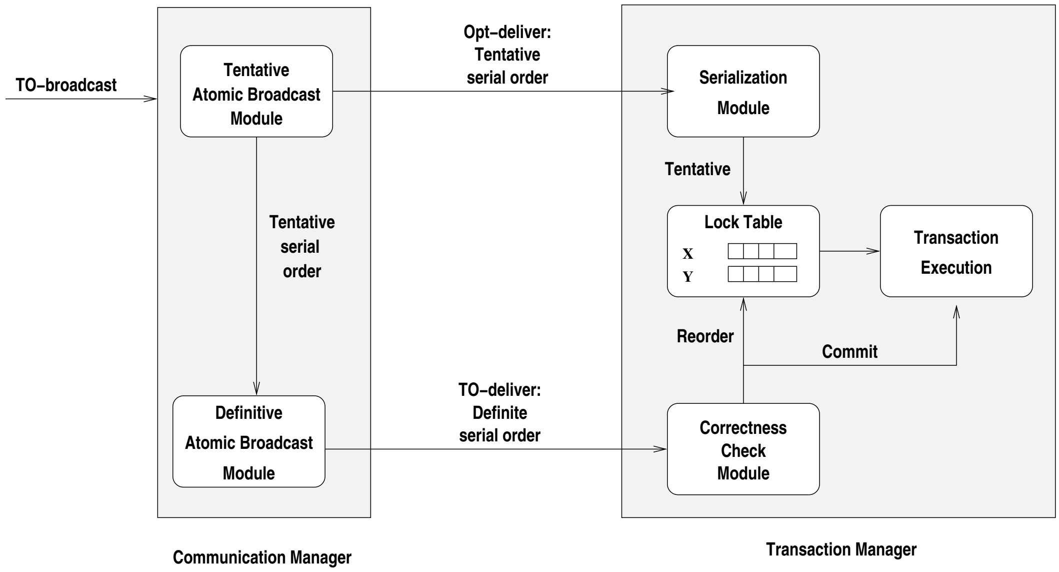

Figure 1: Execution model

corresponding transactions and acquires the corresponding locks. The *Transaction Execution* module executes the operations concurrently as long as they do not conflict. However, whenever they conflict, they are ordered according to the tentative order. If two operations o_i of T_i and o_j of T_j conflict, and T_i is tentatively ordered before T_j, then o_j has to wait until T_i commits before it can start executing. Transactions are not committed until they are `TO-delivered` and their definitive order is determined. Once the communication manager, via the *Definitive Atomic Broadcast* module, establishes a definitive total order for a message, the message is `TO-delivered` to the *Correctness Check* module of the transaction manager. This module compares the tentative serialization order with the serialization order derived from the definitive total order. If they match, then the `TO-delivered` transaction can be committed. If there are mismatches, then measures need to be taken to ensure that the execution order is correct. This may involve, as it will be later discussed, aborting and rescheduling transactions. Using this mechanism, the system guarantees one-copy serializability for the committed transactions.

3 Atomic Broadcast with Optimistic Delivery

In this section we discuss the spontaneous total order property, define the properties of the atomic broadcast primitives on which our database algorithm is based, and discuss some degrees of optimism exploited by atomic broadcast protocols.

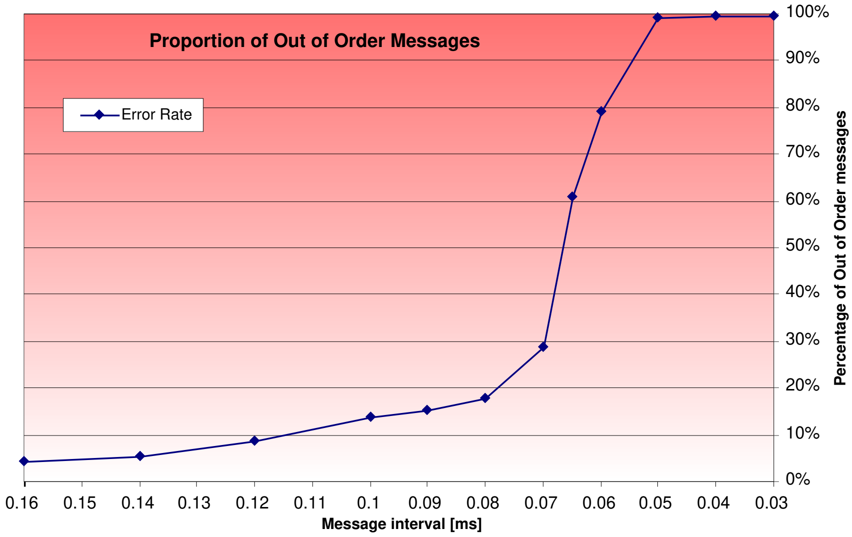

Figure 2: Spontaneous total order in a 11-site-system

3.1 The Spontaneous Total Order Property

Although there exist many different ways to implement total order delivery [6, 4, 7, 18, 27], all of them require some coordination between sites to guarantee that all messages are delivered in the same order at the different sites. However, when network broadcast (e.g., IP-multicast) is used, there is a high probability that messages arrive at all sites spontaneously totally ordered [22]. If this happens most of the time, it seems to be a waste of resources to delay the delivery of a message until the sites agree to this same total order. One could, for example, optimistically process the messages as they are received. If a message is processed out of order, one has to pay the penalty of having to undo the processing done for the message, and redo it again in the proper order. This approach is conceptually similar to the notion of virtual time proposed by Jefferson [13].

To illustrate the spontaneous total order property, we have conducted some experiments in a cluster of eleven sites. Sites are equipped with a Pentium III/766 MHz processor with 128 MB of RAM and a 100-Base TX full duplex Ethernet network interface. The network is built around one hub. The experiment was done using the Neko Framework [26]. In the experiment (see Figure 2) each site broadcasts a messages to all the other sites, and receives messages from all sites over a certain period of time (around 20 sec.). Broadcasts are implemented with IP-multicast, and messages have a length of 98 bytes. Figure 2 shows the percentage of messages that where not spontaneously ordered vs. the interval between two successive broadcasts on each site. For example, for this configuration, if each site sends one message each 0.16 milliseconds, around 95% of the messages arrive at all sites in the same order.

3.2 Degrees of Optimism

Atomic broadcast protocols have traditionally been defined by a single delivery primitive [6, 4, 7, 18, 27] which guarantees that no site delivers a message out of order (total order properties defined in [12]). Only recently, optimistic protocols that exploit the characteristics of the network, or the semantics of the application, have been considered. In [22], the authors propose an Optimistic Atomic Broadcast protocol that first checks whether the order in which messages are received is the same at all sites. If so, the algorithm does not incur in any further coordination between sites to reach an agreement on the order of such messages. Since the verification phase introduces some additional messages in the protocol, there is a tradeoff between *optimistic* and *conservative* (non-optimistic) decisions. However, messages are never delivered in the wrong order to the application (see Figure 3).

The approach proposed here is a more *aggressive* version of the protocol in [22], in that it shortcuts the verification phase. This is possible because the application, that is, the database, allows mistakes (due to optimistic delivery) to be *corrected* by undoing operations and redoing them later, in the definitive order. This approach has significant potential since it does not only rely on the optimism about the network, but also on the semantics of the application, that in this case does not always require messages to be totally ordered at all sites (i.e., if two messages contain transactions that do not conflict, total order between these messages is not necessary).

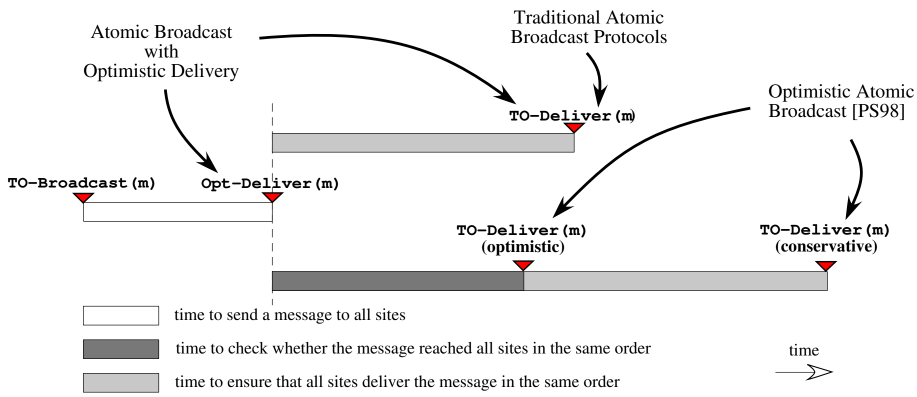

Figure 3: Degrees of optimism

These different degrees of optimism are summarized in Figure 3 (for simplicity, we consider a scenario without failures). Although Figure 3 is not in any scale, it is reasonable to assume that the time to send a message to all destinations is shorter than the time to check whether messages reach their destinations in the same order, which is shorter than the time necessary to order messages. By delivering messages twice, atomic broadcast with optimistic delivery is a very aggressive way of taking advantage of the spontaneous total order property. However, it can only be used if the application (in the case, the database) is able to undo operations that were executed in the wrong order.

The Atomic Broadcast with Optimistic Delivery is specified by the following properties.

Termination: If a site `TO-broadcasts` m, then it eventually `Opt-delivers` m.

Global Agreement: If a site `Opt-delivers` m then every site eventually `Opt-delivers` m.

Local Agreement: If a site `Opt-delivers` m then it eventually `TO-delivers` m.

Global Order: If two sites N_i and N_j `TO-deliver` two messages m and m', then N_i `TO-delivers` m before it `TO-delivers` m' if and only if N_j `TO-delivers` m before it `TO-delivers` m'.

Local Order: A site first `Opt-delivers` m and then `TO-delivers` m.

These properties state that every message `TO-broadcast` by a site is eventually `Opt-delivered` and `TO-delivered` by every site in the system. The order properties guarantee that no site `TO-delivers` a message before `Opt-delivering` it, and every message is `TO-delivered` (but not necessarily `Opt-delivered`) in the same order by all sites.

4 Optimistic Transaction Processing

In this section, we present the basic idea behind optimistic transaction processing. In order to extract the key points we assume that there are only update transactions and each update transaction consists of exactly one write operation. In Sections 5 and 6 we generalize the algorithm to work with arbitrary transactions.

4.1 General Idea

To better understand the algorithms described below, the idea is first elaborated using an example. Assume two sites N and N' where the following tentative sequence of update transactions (messages) is delivered to the database.

$$Tentative\ total\ order\ at\ N:\ T_1,\ T_2,\ T_3,\ T_4$$
$$Tentative\ total\ order\ at\ N':\ T_2,\ T_1,\ T_4,\ T_3$$

Further assume that there are three objects X, Y, and Z. T_1 accesses X, T_2 accesses Y, and T_3 and T_4 access Z. When the scheduler receives the transactions in tentative order, it places them as follows into the lock table:

$$\begin{aligned} At\ N:\quad & lock\ queue\ (X) = && T_1 \\ & lock\ queue\ (Y) = && T_2 \\ & lock\ queue\ (Z) = && T_3,\ T_4 \end{aligned}$$

$$\begin{aligned} At\ \ N': \quad lock\ queue\ (X) = &\quad T_1 \\ lock\ queue\ (Y) = &\quad T_2 \\ lock\ queue\ (Z) = &\quad T_4,\ T_3 \end{aligned}$$

The transaction manager will then submit the execution of the transactions at the head of each queue, i.e., T_1, T_2, and T_3 are executed at N, and T_1, T_2, and T_4 are executed at N'. When the transactions terminate execution, the transaction manager will wait to commit them until their ordering is confirmed. Assume that the definitive total order turns out to be:

$$Definitive\ total\ order : T_1,\ T_2,\ T_3,\ T_4$$

This means that at N, the definitive total order is identical to the tentative order, while at N' the definitive order has changed in regard to the tentative order for transactions T_1 and T_2 and for transactions T_3 and T_4.

Upon receiving the messages in definitive total order, the transaction manager has to check whether what it did make sense. At N, the tentative order and the definitive order are the same, thus, transactions T_1, T_2, and T_3 can be committed, and T_4 can execute once T_3 has committed.

At N', however, things are more complicated since the definitive total order is not the same as the tentative order. However, we can see that the ordering between T_1 and T_2 is not really important because these two transactions do not conflict. However, the order between T_3 and T_4 is relevant since they conflict. Given that the serialization order must match the definitive total order of the communication system in the case of conflicts, the transaction manager has to undo the modifications of T_4 and first perform the ones of T_3 before it re-executes T_4.

It is easy for the transaction manager of N' to detect such conflicts. Assume T_4 and T_3 have been `Opt-delivered` and T_4 is ordered before T_3 in the lock queue for Z. When T_3 is `TO-delivered` (note, that T_3 is `TO-delivered` before T_4), the transaction manager of N' performs a correctness check. It looks into the lock queue of Z and scans through the list of transactions. The first transaction is T_4 and T_4 has not yet been `TO-delivered`. The wrong order is detected and the updates of T_4 can be undone using traditional recovery techniques [3]. T_4 will then be appended to the queue after T_3. To be able to detect whether a transaction in the queue has already been `TO-delivered`, the transaction manager should mark transactions as `TO-delivered`. This can be done during the correctness check. In our example, the transaction manager marks T_3 `TO-delivered` when it performs the check. When at a later time-point T_4 is `TO-delivered`, the transaction manager performs again a correctness check. It looks in the queue and scans through the list of transactions. The first transaction is now T_3. Since T_3 is marked `TO-delivered` the transaction manager knows that this time the scheduling of T_3 before T_4 was correct and no rescheduling has to take place. Thus, T_4 is simply marked `TO-delivered`.

This example shows both the basic mechanisms used as well as how communication and transaction execution can be overlapped and performed at the same time. Note that, whenever transactions do not conflict, the discrepancy between the tentative and the definitive orders does not lead to additional overhead because ordering these transactions is not necessary (see T_1 and T_2 at N'). This means that in the case of low to medium conflict rates among transactions, the tentative and the definitive order might differ considerably without leading to high abort rates (due to reordering).

4.2 The OTP-Algorithm

In the following, we present the OTP-algorithm for optimistic transaction processing. Its main tasks are the maintenance of a serialization order, a controlled execution and a correct termination (commit/abort) of the transactions. For simplicity, we divide the algorithm into different parts according to the different modules described in Section 2.4.

The transaction management relies on the semantics of the primitives `Opt-deliver(`m`)` and `TO-deliver(`m`)` provided by the communication system (see Section 2.1). The serialization module determines the serialization order on behalf of `Opt-delivered` messages, the correctness check module checks and corrects this order on behalf of `TO-delivered` messages, and the execution module executes the transactions. Note that these different modules do not necessarily represent different threads of execution but rather separate the different steps in the lifetime of a transaction. Since all modules access the same common data structures, some form of concurrency control between the modules is necessary (for instance, by using semaphores). Moreover, we assume without further discussing them here that there are two functions, commit and abort, that perform all the operations necessary to commit or abort a transaction locally.

Care must be taken that there is at most one transaction accessing a given object at any time, and that transactions do not commit before they are both executed and `TO-delivered` to guarantee that the serialization order obeys the definitive total order. To do so, we label each transaction with two state variables. The *execution state* of a transaction can be `active` or `executed`. The *delivery state* can be `pending` (after `Opt-deliver`) or `committable` (after `TO-deliver`).

The serialization module is activated upon `Opt-delivery` of a transaction. Its job, depicted in Figure 4, is to append an `Opt-delivered` transaction to the lock queue of the object it is accessing (S1), to mark that this serialization order is still tentative (S2), and to submit the execution of the transaction when there are no conflicts (S4).

The execution module has to inform the transaction manager about completely executed

Upon `Opt-delivery` *of message m containing transaction T_i with operation $w_i(X)$:*

S1 Append T_i to the lock queue X
S2 Mark T_i as `pending` and `active`
S3 **if** T_i is the only transaction in the lock queue
S4   Submit the execution of the transaction
S5 **end if**

Figure 4: OTP-Algorithm: Serialization Module

Upon complete execution of transaction T_i with operation $w_i(X)$:

E1 **if** T_i is marked `committable` (see correctness check module)
E2   commit T_i and remove T_i's lock from the lock queue X
E3   Submit the execution of the next transaction in the queue
E4 **else**
E5   Mark T_i `executed`
E6 **end if**

Figure 5: OTP-Algorithm: Execution Module

transactions (Figure 5). When a transaction is both executed and `TO-delivered` (E1), it can commit (E2). If a transaction has completely executed before its `TO-delivery`, it must be marked accordingly (E5). Note that only the first transaction in a lock queue can be marked `executed`.

The correctness check module is activated upon `TO-delivery` of a transaction. Figure 6 depicts the different steps. The module verifies whether the preliminary execution of a transaction was correct and reschedules the transaction if this is not the case.

Since each message is `Opt-delivered` before it is `TO-delivered` (Local Order property), it is guaranteed that there is an entry for a transaction T in its corresponding lock queue (CC1). The first transaction of a lock queue commits whenever it is `TO-delivered` and totally executed (CC2,CC3) (both events must be true) and the execution of the next transaction in the lock queue can be submitted (CC4). If a transaction cannot be committed immediately upon its `TO-delivery` it is marked `committable` (CC6) to distinguish between transactions whose final serialization order has been determined and those where `TO-delivery` is still pending. The last part of the protocol checks whether the tentative and the definitive order are different for conflicting transactions. If so, abort (CC7,CC8) and reordering (CC10) take place. Note that abort does not mean that the aborted transaction will never be executed and committed. The aborted transaction will be re-executed at a later point in time. The protocol guarantees that all `committable` transactions are ordered before all pending ones in any lock queue (due to step CC10). In particular, if transaction T in the lock queue of X is `TO-delivered` and the first transaction in the queue is still pending, all transactions before T are

Upon `TO-delivery` *of message m containing transaction T_i with write operation $w_i(X)$:*

CC1 Look for the entry of T_i in the lock queue of X
CC2 **if** T_i is marked `executed` (can only be the first one in the queue)
CC3 Commit T_i and remove it from the lock queue
CC4 Submit the execution of the next transaction in the queue if existing
CC5 **else** (not yet fully executed or not the first one in the queue)
CC6 Mark T_i `committable`
CC7 **if** the first transaction T_j in the lock queue is marked `pending`
CC8 Abort T_j
CC9 **end if**
CC10 Schedule T_i before the first transaction T_k in the queue marked `pending`
CC11 **if** T_i has now become the first transaction in queue
CC12 Submit the execution of T_i
CC13 **end if**
CC14 **end if**

Figure 6: OTP-Algorithm: Correctness Check Module

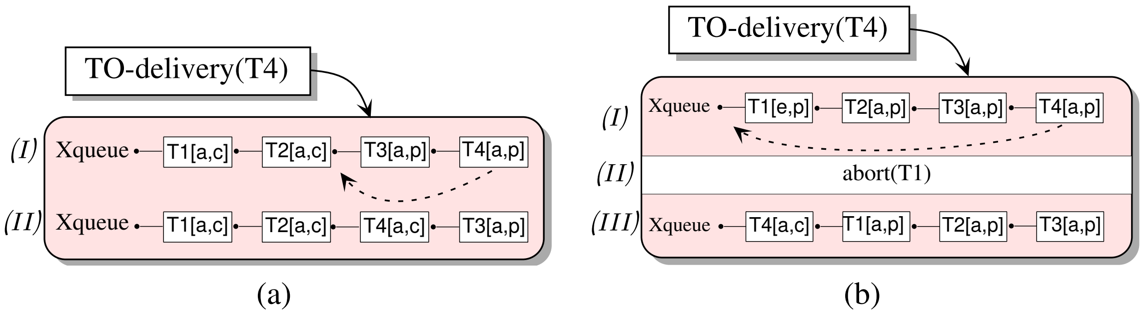

Figure 7: OTP-Algorithm: Examples of reordering transactions: (a) without and (b) with abort

pending. Therefore, step CC10 schedules T to be the first transaction in the queue (CC11), and step CC12 keeps the execution of transactions in this queue running.

We illustrate this further with two examples (see Figure 7). In the following we use the following notation: a for `active`, e for `executed`, p for `pending` and c for `committable`. In the first example, a lock queue for object X has the following entries: $T_1[a, c],\ T_2[a, c],\ T_3[a, p],\ T_4[a, p]$. This means that both T_1 and T_2 have been `TO-delivered`, but not T_3 and T_4 and the execution of T_1 is still in progress. When the `TO-delivery` of T_4 is now processed, T_4 is simply rescheduled between T_2 and T_3 (CC10). Since, the first transaction in the queue, T_1, is committable (it only waits for its execution to terminate) it will not be aborted.

In the second example, the queue for X has the entries: $T_1[e, p],\ T_2[a, p],\ T_3[a, p],\ T_4[a, p]$. This means that none of the transactions is `TO-delivered` but T_1 is already fully executed. In this case, when the `TO-delivery` of T_4 is processed, the first transaction T_1 must be aborted since it is still pending (CC7-CC8). After this, T_4 can be rescheduled before T_1 and submitted. This means that the execution of T_1 is rescheduled after the execution of T_4. These examples show how committable transactions get always ordered before all pending ones.

4.3 Failures and Recovery

The global agreement and global order properties of the Atomic Broadcast with Optimistic Delivery ensure a natural approach to deal with site failures and recoveries. Global order does not allow, for example, failing sites to TO-deliver messages in the wrong order. This means that once the site recovers, the order in which it committed its transactions is consistent with the order other sites committed the same transactions.

Recovering sites have to catch up with operational sites before they are able to perform their normal computation. This notion of recovery is captured by the global agreement property of Atomic Broadcast with Optimistic Delivery, since all sites are supposed to TO-deliver the same transactions. Therefore, if a site cannot TO-deliver a transaction because it has failed, it will do that upon recovering.

Even though such an approach hides the complexities related to site recovery, it may not be efficient if the recovering site has missed ”too many” transactions: the cost of catching up with the operational sites by processing missing transactions may become higher than simply copying the database from an up-to-date site. Coming up with efficient ways to implement site recovery requires a discussion of its own and we do not further address the issue in this paper. For further reference see [15].

4.4 Correctness of the OTP-Algorithm

In this section we prove that the OTP-algorithm is starvation free and provides 1-copy-serializability. Starvation free means that a transaction that is `TO-delivered` will eventually be committed and not rescheduled forever. We use the following notation: for two transactions T_i and T_j accessing both object X, we write $T_i \rightarrow_{Opt} T_j$ if T_i is `Opt-delivered` before T_j. Similarly, we write $T_i \rightarrow_{TO} T_j$ if T_i is `TO-delivered` before T_j.

For Theorem 4.1 we assume a failure free execution.

Theorem 4.1 *The OTP-algorithm guarantees that each* `TO-delivered` *transaction* T_i *eventually commits.*

Proof

We prove the theorem by induction on the position n of T_i in the corresponding lock queue of object X.

1. *Induction Basis:* If T_i is the first transaction in the queue ($n = 1$), it is executed immediately (S3-S4,E3,CC4,CC11-CC12) and commits after its execution (E1-E2,CC2-CC3).

2. *Induction Hypothesis:* The theorem holds for all `TO-delivered` transactions that are at positions $n \leq k$, for some $k \geq 1$, in the lock queue, i.e., all transactions that have at most n-1 preceding transactions will eventually commit.
3. *Induction Step:* Assume now, a transaction T_i is at position $n = k+1$ when the correctness check module processes T_i's `TO-delivered` message. Let T_j be any of the transactions ordered before T_i in the lock queue. Two cases can be distinguished:
   a.) $T_i \rightarrow_{TO} T_j$: When the correctness check module processes the `TO-delivery` of T_i, T_j is still pending. This means, step CC10 will schedule T_i before T_j, and hence, to a position $n' \leq k$. Therefore, according to the induction hypothesis, T_i will eventually commit. Note that due to the reordering process T_j might be moved out of the first k positions. Since it has not yet been `TO-delivered` this does not violate the induction hypothesis.
   b.) $T_j \rightarrow_{TO} T_i$: Since T_j has a position $n' \leq k$, the induction hypothesis assures that T_j will eventually commit and be removed from the lock queue. When this happens, T_i is at most at position k, and hence, will eventually commit according to the induction hypothesis. □

Lemma 4.1 *Each site executes and orders conflicting transactions in the definitive order established by the atomic broadcast.*

Proof

Let T_i and T_j be two conflicting transactions accessing both the same object X and and let $T_i \rightarrow_{TO} T_j$. We have to show that T_i commits before T_j. We can distinguish two cases:

1. $T_i \rightarrow_{Opt} T_j$: This means that T_i is included into the lock queue of X before T_j. We have to show that this order can never be reversed and hence, T_i executes and commits before T_j. The only time the order could change according to the protocol is when the correctness check module processes the `TO-delivery` of T_j. However, at that time, T_i is either already executed and committed or it is marked `committable`, because of $T_i \rightarrow_{TO} T_j$. Hence, CC10 does not affect T_i.
2. $T_j \rightarrow_{Opt} T_i$: This means that T_j is included into the lock queue before T_i. We show that this order is reversed exactly once and hence, T_i commits before T_j. When T_i is `TO-delivered`, T_j might already be executed (when it is the first transaction in the queue) but cannot be committed because it is not yet `TO-delivered` but still marked as pending. Therefore, the protocol processes step CC10 and reorders T_i before T_j. This order cannot be changed anymore because T_i is now marked `committable`. □

Theorem 4.2 *The OTP-algorithm provides 1-copy-serializability.*

Proof

Since up to now, we have only looked at update transactions that are executed at all sites, the local histories of all sites contain exactly the same transactions. The local histories are trivially serial since each transaction only consists of one operation. Lemma 5.1 proves that in all these histories conflicting operations are always processed in the same order, namely the definitive order established by the atomic broadcast. With this, the serialization graphs of the local histories are all identical. Hence, the union of these graphs is identical to each individual graph and acyclic. □

5 Update Transactions with Several Operations

We now move from the simple case of one-operation transactions to the general case in which transactions are allowed to access arbitrary objects of the database.

Our model still uses stored procedures. Whenever a client sends the request for an update transaction to a site N, N `TO-broadcasts` the request to all sites and each site executes the corresponding stored procedure. When a transaction is `OPT-delivered`, the lock requests for all its operations are placed in an atomic step into the corresponding lock queues without interleaving with other transactions. Note that only requesting the locks is performed serially, the execution of the transactions can be done concurrently (of course only, if they do not conflict). This means, however, that all locks of a transaction must be known in advance. We are aware that this is still restrictive but we believe this is acceptable and feasible in the case of predefined stored procedures where one can tell in advance which objects are accessed by the transaction.

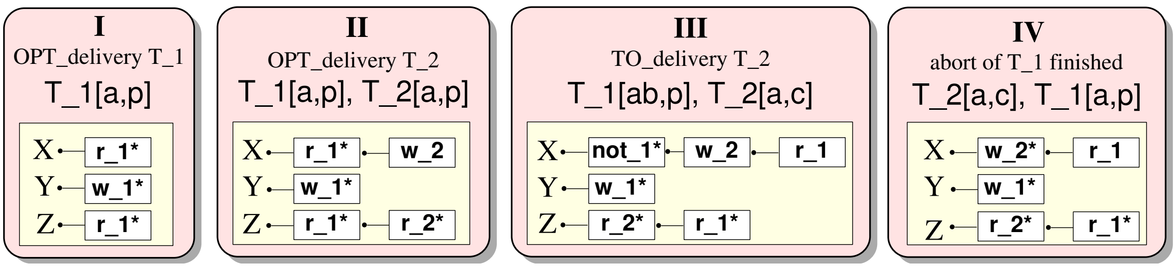

Figure 8: Locking for transactions with several objects

5.1 Example

Figure 8 depicts an example of the enhanced system. We look at a lock table with lock entries for three objects X, Y and Z and two transactions T_1 and T_2. T_1 reads objects X and Z and writes Y. T_2 reads object Z and writes X. Hence, the accesses on X conflict. Steps (I) to (IV) depict different states of the lock table if the following delivery order of messages takes place:

$$Tentative\ total\ order:\ T_1,\ T_2 \qquad\qquad Definitive\ total\ order:\ T_2,\ T_1$$

In step (I) T_1 is `OPT-delivered`. Since there are no other locks active, all locks can be granted and T_1 can start executing. Granted locks are labeled with a star. T_1's state is `active` and `pending`. When T_2 is `OPT-delivered` (II), its lock on Z can be granted (because both T_1's and T_2's locks are reads) and the operation submitted, but its write lock on X has to wait until T_1 releases its lock. Both transactions are `active` and `pending`. Step (III) depicts the `TO-delivery` of T_2. We scan through all of T_2's locks and look whether there exist locks of `pending` transactions that are ordered before the waiting locks of T_2. Note that all locks of a single transaction are usually linked with each other making this check a fast operation. In our example, `pending` transaction T_1 has a conflicting granted lock on X and thus, must be aborted. Only when the updates of T_1 are undone, the lock can be granted to T_2. However, this might take considerable time (because T_1 might already have executed many operations). Since we do not want to wait, we reorder immediately T_2's locks before T_1's locks. Additionally, we keep a `notification entry` at the beginning of the queue for X. It can be viewed as a copy of T_1's lock entry. When T_1's abort is completed it removes the notification entry from the queue and only then T_2's lock can be granted. Such a notification entry on an object is only created if the lock of the `pending` transaction and of the `committable` transaction conflict on this object because only in this case the `committable` transaction must wait. Hence, there is no notification entry for Z (shared locks) or Y (T_2 does not need a lock for Y). Step (IV) depicts the time after T_1 is aborted. At the end of its abort, T_1 scans through all its locks and whenever it finds one of its notification entries (in our case X) it releases the entry so that the waiting requests can be granted. Note, that its original lock entry on X has already been queued behind the `TO-delivered` entry of T_2 in step (III). Then, T_1 is restarted from the beginning and its execution state changes from `aborting` back to `active`. The `TO-delivery` of T_1 does not change anything. When performing the check, only locks of the `committable` transaction T_2 are ordered before T_1. Hence, T_1 is simply marked `committable`. Whenever both transactions are fully executed their locks can be released and the transactions can be committed.

5.2 G-Algorithm

In this section, the enhanced algorithm, called G-algorithm (general algorithm) is described. As before, we divide transaction execution into the steps serialization, execution and correctness check.

Upon Opt-delivery *of message m containing transaction T_i:*
S1 **for each** operation $o_i(X)$ of T_i:
S2 Append a lock entry in queue X
S3 **end for each**
S4 Mark T_i `pending` and `active`
S5 Submit the execution of the transaction (see execution module E1-E4)

Figure 9: G-Algorithm: Serialization module

Upon submission to execute transaction T_i:
E1 Mark T_i `active`
E2 **for each** operation $o_i(X)$:
E3 As soon as the corresponding lock is granted
E4 execute $o_i(X)$
E5 **end for each**
Upon complete execution of transaction T_i:
E6 **if** T_i is marked `committable` (see correctness check module CC6)
E7 Commit T_i
E8 **for each** lock on object X
E9 Release the lock
E10 **end for each**
E11 **else**
E12 Mark T_i `executed`
E13 **end if**
Upon submission to abort transaction T_i: (by CC11 of Correctness Check Module)
E14 Mark T_i `aborting`
E15 Undo all operations executed so far
Upon complete abort of transaction T_i:
E16 **for each** `notification entry` on object X (see correctness check module)
E17 Release the entry
E18 **end for each**
E19 Restart the execution (E1-E4)

Figure 10: G-Algorithm: Execution Module

Figure 9 depicts the serialization module. Upon `Opt-delivery` of a transaction, all lock entries are created and included into the lock table (S1-S3). Some might be granted immediately, others might have to wait. Entering all locks into the lock table is considered one atomic step, i.e., the process may not be interrupted by other accesses to the lock table. As noted before this can be done by using a latch on the lock table. At this stage, the transaction has the delivery state `pending` (S4). Only after the locks are requested the transaction starts executing (S5).

The execution module (Figure 10) is responsible for processing transactions. Here, we distinguish several cases: a transaction is submitted for execution, a transaction has completely executed, the abort of a transaction is submitted, and a transaction has completely aborted. A transaction can now be in three different execution states. It can be `active`, `executed` or `aborting`. Once a transaction is submitted by the serialization module it is `active`

Upon `TO-delivery` *of message m containing transaction T_i:*
CC1 **if** T_i is marked `executed` (see execution module E12)
CC2 Commit T_i
CC3 **for each** lock entry on object X
CC4 Release the lock
CC5 **end for each**
CC6 **else**
CC7 Mark T_i `committable`
CC8 **for each** lock entry X of T_i
CC9 **for each** conflicting granted lock of a `pending` transaction T_j
CC10 **if** T_j is not marked `aborting`
CC11 start aborting T_j (see execution module E14-E15)
CC12 **end if**
CC13 **end for each**
CC14 Schedule T_i's lock before the first lock entry
CC15 that belongs to a `pending` transaction T_k.
CC16 **for each** conflicting granted lock of a `pending` transaction T_j
CC17 Keep a `notification entry` at the begin of the queue.
CC18 Only when this entry is released (see execution module E16-E18),
CC19 T_i's lock can be granted
CC20 **end for each**
CC21 **end for each**
CC22 **end if**

Figure 11: G-Algorithm: Correctness Check Module

(E1). In our protocol, all locks are requested at the beginning of the transaction. Then the stored procedure is started executing the sequence of read and write operations as soon as the corresponding locks are granted (E2-E4).

Once the transaction is fully executed we look whether the transaction has already the delivery state `committable`. If this is the case we can commit it and release its locks (E6-E10). Otherwise, the transaction transfers from the execution state `active` to `executed`.

As long as the transaction is in the delivery state `pending` it might happen that it has to abort (E14-E15) due to a mismatch between the `OTP-delivery` and the `TO-delivery` order (CC11 of correctness check module, Figure 11). Note that it can transfer both from the `active` and the `executed` execution state into the `aborting` state.

Once the transaction is completely aborted it releases all its notification entries (E16-E18). These entries were created during the correctness check. They replaced locks that conflicted with locks of `committable` transactions. For details of how these entries are created see the correctness check module. Finally, the transaction is restarted (E19). Note that the restart transfers the transaction from the execution state `aborting` back to the `active` state (E1).

The correctness check module (Figure 11) is activated upon `TO-delivery` of a transaction T_i. As the serialization module, the correctness check module performs its access to the lock table in an atomic step, i.e., it has exclusive access to the lock table until the check is completed. A `TO-delivered` transaction can immediately commit and release its lock if it is already totally executed (CC1-CC5). If this is not the case we mark T_i `committable` (CC7) and check whether any reordering has to take place. We have to reorder entries whenever a lock of a `pending` transaction T_j is ordered before one of T_i's locks. Furthermore, if the lock of T_j is granted and conflicts with T_i's lock, T_j must be aborted to guarantee that conflicting operations are ordered in the order of `TO-delivery`. Hence, the correctness check scans through all locks of T_i (CC8). Whenever there exists a conflicting granted lock of a `pending` transaction T_j, T_j must be aborted. Note that T_j might already be in the `aborting` state if there was already another conflicting transaction `TO-delivered` before T_j. Only if this is not the case we have to submit T_j's abort (CC10-CC12). Then, rescheduling takes place. T_i's lock entry is ordered before the first lock that belongs to a `pending` transaction (CC14-CC15). Note that rescheduling takes place both for conflicting and non-conflicting locks. Since we want to finish the correctness check before the aborts complete, we keep notification entries at the begin of the queue for each conflicting granted lock of a `pending` transaction (CC16-CC20). Only when these locks are released (namely, when the corresponding transactions have completely aborted – see execution module E16-E18) T_i's locks can be granted.

Note that execution state and delivery state are orthogonal to each other. Figure 12 depicts all possible states of a transactions and their transitions. A transaction always starts with the states `active` and `pending` and commits with `executed` and `committable`. Delivery states can only change from `pending` to `committable`. Once it is `committable` there is no cycle in the transitions.

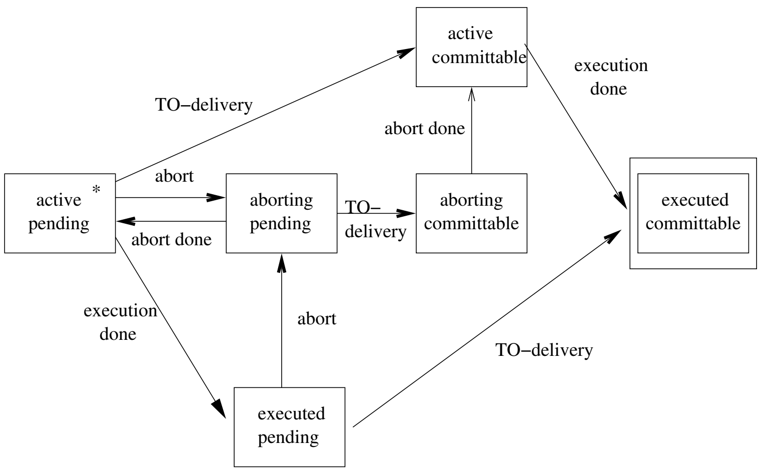

Figure 12: G-Algorithm: Execution and delivery states of transactions

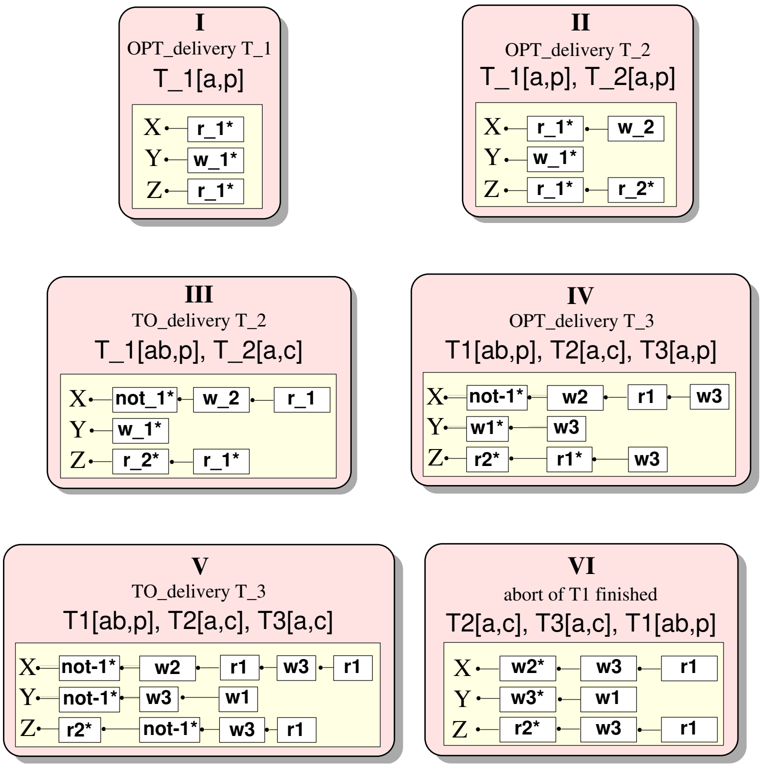

Figure 13: G-Algorithm: Example with reordering

Figure 13 shows a second example with three transactions. The ordering is

$$Tentative\ total\ order:\ T_1,\ T_2,\ T_3$$

$$Definitive\ total\ order:\ T_2,\ T_3,\ T_1$$

The example shows the different steps when the delivery takes place in the following order: `OPT-deliver`(T_1), `OPT-deliver`(T_2), `TO-deliver`(T_2), `OPT-deliver`(T_3), `TO-deliver`(T_3), `TO-deliver`(T_1).

As in the previous example, T_1 reads X and Z, and writes Y, and T_2 writes X and reads Z. Furthermore, T_3 writes X, Y and Z. The first three steps are the same as in the previous example. First (step I), T_1 is `OPT-delivered`, its locks granted and its execution submitted (E1-E5, S1-S5). Its state is `active` and `pending`. When T_2 is `OPT-delivered` (step II) its lock on Z is granted, the lock on X must wait. T_2's state is `active` and `pending`. Upon `TO-delivery` of T_2 (step III), T_1 must be aborted due to the conflict on X (CC9-CC13, E14-E15). T_2's locks on X and Z are scheduled before T_1's locks (CC14-CC15). Note that lock entries are rescheduled both for conflicting locks (X) and non-conflicting locks (Z). Furthermore, T_1 keeps a notification entry on X until it is totally aborted (CC16-CC20). Now, T_1 is `aborting` and `pending` while T_2 is `active` and `committable`. In step IV, T_3 gets `Opt-delivered`. The lock entries are simply added to the queues. They all must wait and T_3 is `active` and `pending`. Next, T_3 is `TO-delivered` (step V). The correctness check scans through all of T_3 locks and finds conflicting granted locks of `pending` transaction T_1 on the objects Y and Z (CC9). Note that T_1's lock on X is no more granted and already reordered behind T_2' lock. Since T_1 is already in the `aborting` state, steps CC10-CC12 of the correctness check algorithm are not performed. T_3 simply orders its locks before T_1's lock entries (CC14-CC15). Furthermore, notification entries for T_1 on objects Y and Z are created. Since T_1 has already a notification entry on X, there is no need to create a second one (CC16-CC20). T_3 is now `active` and `committable`. At this stage, all locks are in the correct order of `TO-delivery`. Additionally there are some notification entries that control when locks are granted. When T_1 has completely aborted (step VI), it releases its notification entries and restarts (E16-E19). The `TO-delivery` of T_1 does not change anything. Whenever one of the three transactions is fully executed it can be committed, its locks are released and granted to the next waiting transaction.

5.3 Proof of Correctness

The proof of correctness follows the same line as the one provided in section 4.4. First we want to show that 1-copy-serializability is provided.

Lemma 5.1 *Using the G-algorithm, each site orders and executes conflicting operations in the definitive order provided by the atomic broadcast.*

Proof

Let T_i and T_j be two conflicting transactions accessing both object X and let $T_i \rightarrow_{TO} T_j$. We have to show that T_i's operation on X is executed before T_j's operation. The proof is identical to the one provided in section 4.4. If $T_i \rightarrow_{OPT} T_j$ the locks were already included in the queue in the right order. This order will not change anymore since the correctness module only reschedules entries if there is a mismatch between the `OPT-delivery` and the `TO-delivery`. Hence T_i's lock will be granted before T_j's locks resulting in the correct order of execution. If $T_j \rightarrow_{OPT} T_i$, then the the `TO-delivery` of T_i will schedule T_i's lock before T_j's lock. In the case T_j's lock was already granted, T_j will undo its operation and only receive the lock and re-execute the operation when T_i has committed. □

Theorem 5.1 *The G-algorithm provides 1-copy-serializability.*

Proof

Again the proof is very similar to the one provided in section 4.4. All local histories contain exactly the same update transactions. Since all transactions request all their locks in an atomic step, the histories are all conflict-serializable. Since all conflicting operations are executed in the same order at all sites, namely the definitive order of the atomic broadcast, all serialization graphs are identical, and hence, the union of these graphs is acyclic. □

Finally we prove that the protocol is starvation free.

Theorem 5.2 *The G-algorithm guarantees that each* `TO-delivered` *transaction T_i eventually commits.*

Proof

We prove the theorem by induction on the position n of T_i in the definitive total order.

1. *Induction Basis:* If T_i is the first `TO-delivered` transaction the correctness check module will schedule T_i's locks before any other lock. Only notification entries might be ordered before T_i's locks. However, the corresponding transactions abort without acquiring any further locks and hence will finally release their notification entries. Hence, eventually all of T_i's locks will be granted, T_i can execute all its operations and commit.
2. *Induction Hypothesis:* The theorem holds for all `TO-delivered` transactions that are at positions $n \leq k$, for some $k \geq 1$, in the definitive total order, i.e., all transactions that have at most n-1 preceding transactions in the total order will eventually commit.
3. *Induction Step:* Assume now, a transaction T_i is at position $n = k + 1$ of the definitive total order. The correctness check module will schedule all of T_i's locks behind all locks of `committable` transactions and before all locks of `pending` transactions (CC14-CC15). All of these `committable` transactions were `TO-delivered` before T_i and have a lower position in the definitive total order. Hence, they will all commit according to the induction hypothesis. Existing notification entries will finally be released as described before. Therefore, T_i's locks will finally be granted, T_i can execute all its operations and commit. □

Note that in this case, starvation free means not only that a transaction is not rescheduled forever but also that the protocol is deadlock free. Transactions do not wait for each other to release locks while holding locks other transactions are waiting for. This is true because the locks of `committable` transactions are ordered in the definitive total order as noted in the proof above.

6 Queries

A configuration consisting of sites all performing exactly the same update transactions is only useful for fault-tolerance purposes. A more common setting will be a database system where the main load are queries. These queries should be executed locally at a single site in order to provide fast response times. Therefore, a protocol needs to be not only tuned for updating transactions but also for read-only transactions.

There exist many concurrency control approaches for queries [3, 11, 19]. Current solutions include standard 2-phase-locking (no difference between queries and updating transactions), optimized locking protocols, and snapshot mechanisms (which eliminate any interference between queries and updating transactions). Replication control should be able to adapt to these different mechanisms and change concurrency control for queries as little as possible.

In what follows we sketch how three different solutions can be integrated into the replica control presented in this paper. The protocols are extensions of the G-algorithm for updating transactions described in Section 5. Note that all these solutions assume that user requests must be declared as queries or update transactions. Stored procedures support this approach very well. Since they are predefined, the type of the transactions (query or update) can be declared in advance.

The basic idea of the proposed algorithms is to smoothly integrate queries into the serialization order given by the total order. We have already shown before that having only update transactions, each execution in the system is conflict-equivalent to a serial execution of the transactions in the order of their `TO-delivery`. In the following, we denote a transaction as T^i if the transaction was the ith message to be `TO-delivered`. That is, using the G-algorithm all executions are conflict-serializable to the serial execution $T^1, T^2, T^3, ...$ We now extend the G-algorithm such that, although a query is only executed locally, this global serialization order is not reversed.

6.1 Simple Query Execution

The first solution is a simple extension of the G-algorithm and handles queries very similar to update transactions (with the big difference, of course, that they are completely local and do not have any message overhead). The basic idea is to treat the start of a query in the same way as an update transaction at its timepoint of `TO-delivery`. Figure 14 depicts the G-Q-protocol. It shows that upon its start a query Q requests all its locks and these locks are ordered after all already committable and before all pending transactions. If a pending transaction holds a conflicting lock it is aborted and rescheduled after the query Q. Lines Q3-Q14 are basically identical to CC9-CC21 of Figure 11.

Theorem 6.1 *The G-Q-algorithm provides 1-copy-serializability.*

Upon begin of query Q:

Mark Q `committable`
for each operation $o(X)$ of Q
    for each conflicting granted lock on X of a `pending` transaction T_j
        if T_j is not marked `aborting`
            start aborting T_j
        end if
    end for each
    Insert Q's lock for X before the first lock entry
    that belongs to a `pending` transaction T_k.
    for each conflicting granted lock of a `pending` transaction T_j
        Keep a `notification entry` at the begin of the queue.
        Only when this entry is released, Q's lock can be granted
    end for each
end for each

Figure 14: G-Q-protocol for queries

Proof Let T^i be the last transaction `TO-delivered` before the start of query Q. The algorithm orders Q in all lock queues after all transactions T^j with $j \leq i$ and before all transactions T^j with $i < j$. Hence, we can add Q to the serialization order by simply ordering it directly after T^i and before T^{i+1}. □

6.2 Dynamic Query Execution

The disadvantage of the G-Q-protocol is that a query must know all the data it wants to access at start time, since this is the time when it acquires all its locks. This might not be feasible for queries since they often have an ad-hoc character (in contrast to update transactions which are usually short and well defined). Furthermore, queries can be very extensive, accessing many data objects and executing for a long time. Locking all data at the beginning will lead to considerable delay for concurrent update transactions. Hence, it is desirable to handle queries dynamically, i.e., queries should neither need to know in advance which data they are going to access nor should transactions be delayed too long by queries.

However, we cannot simply enhance the G-Q-protocol and allow queries to request their locks whenever they want to access a new object (standard 2-phase locking). Such a protocol would violate 1-copy-serializability. The problem is that update transactions accessing different objects could now be indirectly ordered by queries that access both objects. As an example, assume two objects X and Y, transaction T^1 updating X, and the next transaction T^2 updating Y. A query Q runs at node N and access X after the `TO-delivery` of T^1 and then Y before the `TO-delivery` of T^2. Concurrently, a query Q' executes at node N'. It first accesses X

Change of execution module of the G-algorithm (Figure 10):
Upon complete execution of update transaction T^i:
E6 **if** T^i is marked `committable`
E7 Timestamp each object updated by T^i with i and commit T^i
Change of correctness check module of the G-algorithm (Figure 11):
Upon `TO-delivery` *of message m containing update transaction T^i:*
CC1 **if** T^i is marked `executed`
CC2 Timestamp each object updated by T^i with i and commit T^i
...
CC6 **else**
...
CC8 **for each** lock entry X of T^i
...
CC20.1 **for each** conflicting granted lock of a query Q
CC20.2 set Q's $TS_{before} = min(TS_{before}, i)$
CC20.3 **end for each**
Query Execution:
Upon begin of query Q:
Q1 Set Q's $TS_{after} = 0$ and $TS_{before} = \infty$
Q2 **whenever** an operation $o(X)$ of Q is submitted
Q3-Q14 Identical to Q3-Q14 of the G-Q-algorithm (Figure 14)
Q15 **whenever** a lock on object X with timestamp i is granted
Q16 set Q's $TS_{after} = max(TS_{after}, i)$
Q17 **whenever** $TS_{after} \geq TS_{before}$
Q18 abort Q

Figure 15: G-DQ-protocol for queries

before the `TO-delivery` of T^1. Then T^1 and T^2 are `TO-delivered`. Only after that, Q' requests a lock on Y. This means that Q implicitly builds the serialization order $T^1 \rightarrow Q \rightarrow T^2$ at site N (obeying the order given by the total order multicast), while Q' leads to the undesired order $T^2 \rightarrow Q' \rightarrow T^1$ at N' [2] resulting in a cyclic global dependency graph.

Our approach to avoid this problem is to guarantee that if a query indirectly orders two update transactions, then only it does so according to their `TO-delivery`. With this, each local history remains conflict-serializable to the serial history that executes transactions according to their `TO-delivery`. Our solution, the G-DQ-protocol depicted in Figure 15 is a simple enhancement of the G-algorithm of Section 5 using standard 2-phase-locking for queries and a special timestamp mechanism. The figure only depicts changes and add-ons to the original G-algorithm. For each query, we maintain two timestamps TS_{after} and TS_{before}. At any timepoint these timestamps indicate that the query Q must be serialized $T^{TS_{after}} \rightarrow Q \rightarrow T^{TS_{before}}$. At the start of a query, TS_{after} is set to 0 and TS_{before} is set to ∞ (line Q1). Objects are also timestamped. An object X is timestamped with the index i of the last `TO-delivered` transaction

	Upon begin of query Q:	
Q1	Let T^i be the last transaction `TO-delivered` before Q started	
Q2	**whenever** Q wants to read an object X	
Q3	read version X_j such that $j \leq i$ and there is no X_k such than $j < k \leq i$	

Figure 16: G-SQ-protocol for queries

T^i that updated it. X must receive its timestamp some time before T^i releases its lock (for instance, at E7 and CC2). Whenever a transaction T^i is `TO-delivered` and executes the correctness check module, it checks whether there is a conflicting lock of a query Q granted. If this is the case Q's TS_{before} timestamp is set to $min(TS_{before}, i)$ (CC20.1 - CC20.3). We do this, because T^i will write X after Q has read it, hence, Q must be serialized before T^i. Similarly, whenever a query Q reads an object X with timestamp i, its TS_{after} is set to $max(TS_{after}, i)$ (Q15-Q16). This indicates that Q must be serialized after T^i since Q read an object last written by T^i. With this, we can easily detect when Q will violate the serialization order. After each setting of TS_{before} or TS_{after} of a query Q, we check whether $TS_{after} \geq TS_{before}$. If this is the case, Q must be aborted (Q17-Q18), since this indicates that Q indirectly orders $T^{TS_{after}} \rightarrow Q \rightarrow T^{TS_{before}}$ violating the serialization order given by `TO-delivery`. Taking the example at the beginning of this section, query Q' at node N' will abort when its lock on Y is granted, since TS_{before} is 1 (when T^1 is `TO-delivered`, Q' holds a lock on X), and TS_{after} is set to 2 (Y has timestamp 2 since T^2 was the last one to write Y).

Theorem 6.2 *The G-DQ-algorithm provides 1-copy-serializability.*

Proof

Update transactions alone produce schedules according to the G-algorithm that are conflict-serializable to the serial schedule executing all transactions in `TO-delivery` order. A query can only commit if $TS_{after} < TS_{before}$. In this case it can be serialized anywhere between these $T^{TS_{after}}$ and $T^{TS_{before}}$ without violating the given conflict-serializability. □

6.3 Snapshots

A different approach to the problem is to use snapshots for queries (similar to Oracle snapshots [19]). In snapshot queries, queries do not acquire locks and do not interfere with concurrent update transactions at all. Instead, they receive a snapshot of the data that reflects the state of the database at the time the query starts. This can be accomplished by maintaining multiple versions of each object. That is, whenever a transaction updates an object, a new version is created. Our approach timestamps each version of an object X with index i if transaction

T^i created this version. Figure 16 depicts the G-SQ-protocol using snapshots. Whenever a query Q wants to read an object X, it reads the version X_i such that T^i was `TO-delivered` before Q started, and there is no transaction T^j, such that T^j also updated X, T^j was also `TO-delivered` before Q started, and $i < j$.

Theorem 6.3 *The G-SQ-algorithm provides 1-copy-serializability.*

Proof The proof is nearly identical to the proof for the G-Q-algorithm. In the G-SQ-algorithm, a query Q can be serialized after transaction T^i that was the last one `TO-delivered` before Q started. This is also true for the G-SQ-algorithm since Q receives a snapshot of the data that contains all updates performed up to T^i and none of write operations performed by transactions `TO-delivered` after T^i. □

7 Performance Analysis

We have performed extensive simulation experiments evaluating the impact of various factors on the performance of the G-algorithm, and comparing the G-algorithm with other solutions.

Our comparison includes a traditional replica control technique based on distributed locking. In this approach, a transaction is sent to one site in the system. This local site performs all read operations of the transaction. Write operations are forwarded to all sites in the system and the local site waits for a confirmation from all sites that the lock where acquired before it continues with the next operation. Locking at the different sites is autonomous and each site can determine by itself whether it will execute the operation or deny its execution. As pointed out in [10], such an approach leads to high transaction response times and may lead to many distributed deadlocks.

We also consider an algorithm in which the atomic broadcast with optimistic delivery is replaced by a traditional atomic broadcast primitive (a message is only delivered once in its definite total order without optimistic assumptions). Using traditional atomic broadcast, the concurrency control and serialization modules can be greatly simplified. A transaction is again sent to all sites. Upon its delivery, all locks are included in the lock queues. Whenever a lock is granted, the corresponding operation can be executed. Once all operations have been executed, the transaction commits and releases all its locks. Reordering and aborts do not take place. This technique is very similar to active replication [23]. Comparing our protocol with a protocol based on a traditional atomic broadcast helps us to understand the role played by the optimistic delivery component since both protocols are identical in their other components.

7.1 Settings

We use the simulation package C-Sim[17]. We simulate both the database system and the communication system, using CPU, disks, and network as low level resources. The system models multiple servers and multiple clients; each client connects to a given server. A client submits transactions to its local server. The local server forwards the transaction to all servers in the system where it is executed according to it's algorithm. Once a transaction has committed at the local server, the local server sends a confirmation to the client. The work-cycle of a client is the following: the client produces a transaction, sends it to its server, waits for the transaction to terminate, sleeps during some time and starts the cycle again with a new transaction. The load of the system is determined by the number of clients per server and the interval between transactions sent by the clients.

Databases are modeled as a data manager and a lock manager. The unit of locking is a data item. Locking tables are in memory and their use involves no disk access. Accessing data items involves disks and CPU accesses. A distributed deadlock detection scheme was also implemented for the distributed locking scheme.

The communication system simulates a fault-tolerant atomic broadcast algorithm based on [6]. For the simulations, we only consider the most common case of failure-free and suspicion-free executions of the algorithm. The network is modeled as a shared resource. Exchanging a message implies using the CPU resource on the sender site, then the network, and then the CPU on the receiver node [25].

Parameter	Value
Number of items in the database	1000
Number of Servers	3
Number of Clients per Server	6
Disks per Server	2
CPUs per Server	2
Transaction Length	2 – 6 Operations
Probability that an operation is a write	20%
Time interval between transactions	100 – 325 *ms*

Parameter	Value
Buffer hit ratio	80%
Time for a read	4 – 12 *ms*
Time for a write	4 – 12 *ms*
CPU Time used for an I/O operation	0.4 *ms*
Time for a message on the Network	1 *ms*
CPU time to send/receive a message	1 *ms*
Time for a broadcast on the Network	1 *ms*
CPU time to send/receive a broadcast	1 *ms*
Out of order messages	20%

Table 1: Simulation parameters

Simulations were run until the mean response times was with a probability of 95% within a confidence interval of 1%. The basic parameter settings are given in Table 1. Most of the experiments include three servers, each with six connected clients (so a total of 18 clients connected to the system). The data set contains 1000 items. Servers are composed of two CPUs and two disk units; each CPU has access to both disks, but only one CPU can access a disk at a time. Data items are distributed on the different disks. The transaction length is uniformly distributed between 2 and 6 operations. Each operation is either a read (80%) or a write (20%). The delay between the submission of two consecutive transactions at a client varies between 100 and 325 *ms* by 25 *ms* increments. Small submission intervals depict high workloads, large intervals depict small workloads. Operating a read or write operation uses the disk between 4 and 12 *ms* (uniform distribution). Read operations have a 80% chance of hitting the cache, and therefore occur no disk usage. Each input/output operation (read and write) has a CPU overhead of 0.4 *ms*. Sending a point-to-point message consumes 1 *ms* of CPU at the sender, 1 *ms* of the network resource and 1 *ms* of CPU at the receiver. We assume a low-level multicast facility (like IP-multicast) with which we can send a multicast in a single operation. The cost is 1 *ms* on the network, and 1 *ms* of CPU at both the sender and the receiver. The probability that a message is optimistically delivered in the wrong order is set to 20% in most of the experiments (a very conservative choice, given the data in Figure 2).

7.2 Results

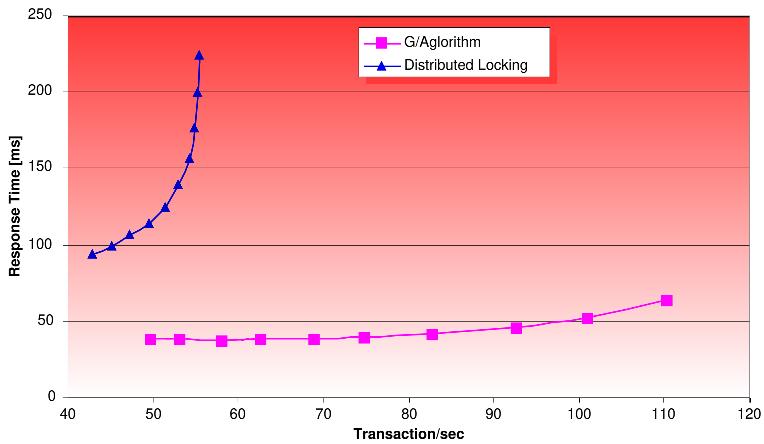

Figure 17: Response times of G-Algorithm vs. Distributed Locking

7.2.1 G-Algorithm vs. Distributed Locking

Figure 17 compares transaction response times of the G-algorithm and traditional distributed locking as a function of the workload. The workload was controlled by changing the time interval between transactions. In general, response times increases with high workloads as resource contention get higher. Comparing both algorithms, distributed locking has much higher response times than the G-algorithm. As mentioned earlier, this was to be expected since distributed locking generates many more messages than the G-algorithm. Also, using distributed

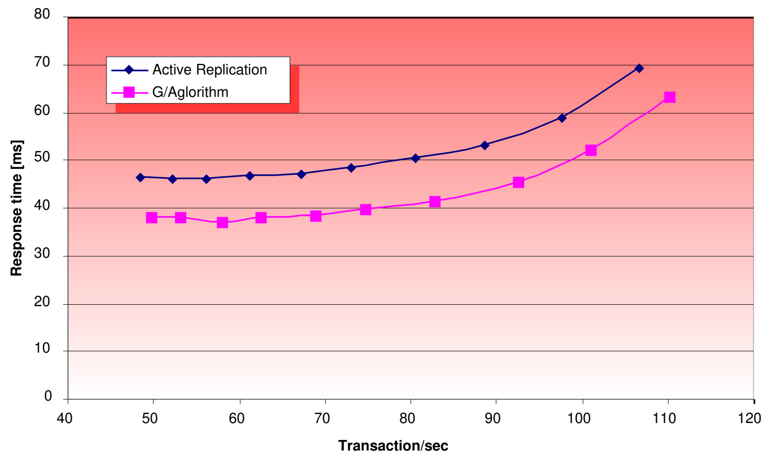

Figure 18: Response times of the G-algorithm and Active Replication

locking, a transaction can only start an operation when the execution of the previous operation has completed at all sites. Using the G-algorithm, a site can commit a transaction once it is locally executed and the total order is determined. Hence, the response time of a transaction does not depend on the execution time at the other sites.

Abort rates do not have a significant impact in this experiment being below one percent for both algorithms.

7.2.2 G-Algorithm vs. Active Replication

Figure 18 compares the response times of the G-algorithm and active replication as a function of the workload. The G-algorithm has smaller response times than the active replication approach throughout the entire test configuration. The performance gain directly results from the optimistic delivery and the overlap of communication and transaction processing. This gain is proportional to the time needed for the communication system to determine the total order. Executing atomic broadcast in the simulator corresponds to executing 3 multicasts and $n-1$ concurent point-to-point messages (n is the number of servers). Using the G-algorithm, processing of the transaction can begin after the first multicast. Therefore the theoretical gain would be the time needed for 2 broadcast and 1 point-to-point message. Given the parameters of the simulator, this gives $2 \times 3 + 1 \times 3 = 9\ ms$. The observed difference was between 8 and 11 *ms*, quite close to the predictions.

Interestingly, the performance gain is slightly smaller in small load situations (around 8 *ms*). One reason for this is that transaction processing time is smaller than message delivery time. Transactions are processed before TO-delivery. As a result, processing is delayed until TO-delivery and a smaller performance gain is observed. Figure 19 shows this phenomenon, where

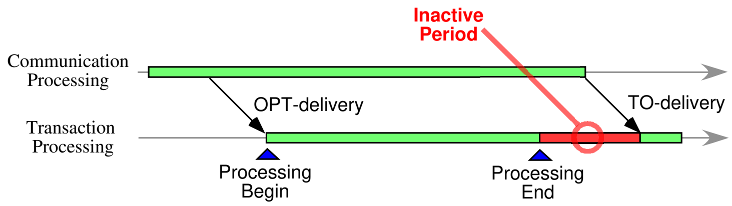

Figure 19: Overlap with long communications

the first bar represents the communication processing, and the second the transaction processing. A message is first `OPT-delivered`, at which point the transaction processing begins. If the message is not yet `TO-delivered` when the transaction processing ends, processing becomes inactive until the message is `TO-delivered`. During this inactive period, locks are held and might cause higher contention.

7.2.3 Size of the System

In this experiment we analyze how the number of replicas influences the performance of the G-Algorithm by varying the number of nodes and the number of clients per node. The workload is the same throughout all tested scenarios. The first scenario is the non replicated case, with one server and 18 clients connected to it – in this case transaction are locally executed without any communication overhead. This scenario depicts the lower bound of what can be achieved in terms of performance. The second scenario consists of two server, with each nine clients attached; the third scenarios consists of three servers with each six clients attached; the fourth scenario consists of six servers with each three attached clients, the last scenario was with nine servers with each two attached clients.

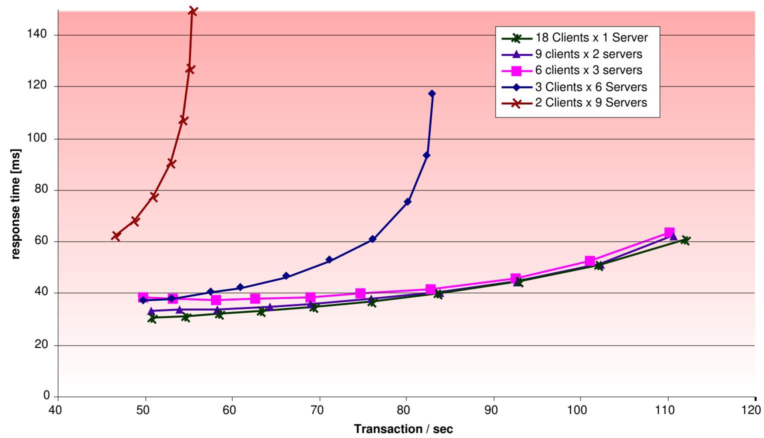

Figure 20: Response times for different server configurations (G-Algorithm)

Figure 20 shows the response times for the five scenarios. Settings with a low number of servers (1,2 or 3) have roughly the same performance. As can be expected, large systems have higher response times than small systems.

The reason for this is the overhead of the atomic broadcast. Since it contains one message round with point-to-point messages where each server sends an acknowledgment to the coordinator node, the protocol does not scale well. This is particularly true for our test configuration which represents a rather slow communication component.

7.2.4 Spontaneous Total Order Property

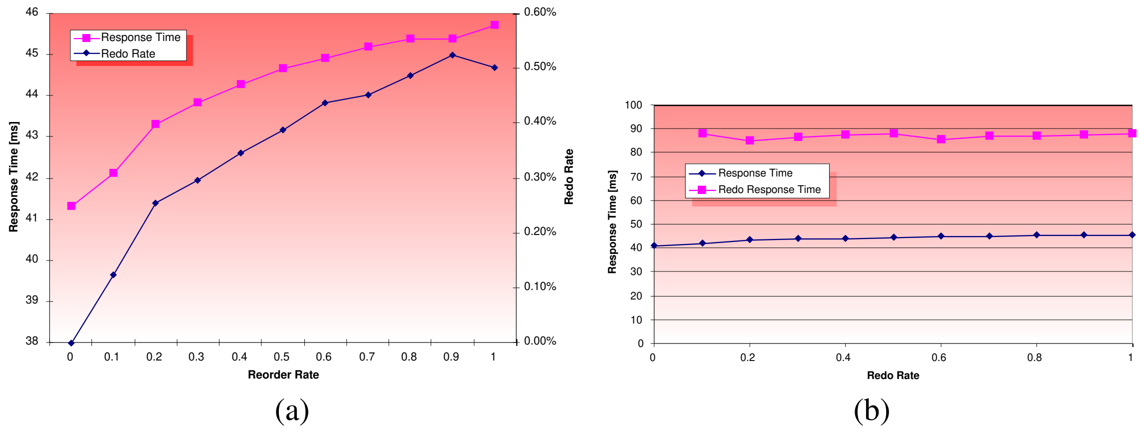

Figure 21: Redo rate as function of out of order message rate (a) and Response times of rescheduled and non-rescheduled transactions (b)

The next experiment evaluates the effects of the spontaneous total order property on the performance of the G-algorithm.

In this test scenario, we fixed the transaction interval at 150 *ms*. The conflict rate at this workload using the settings of Table 1 reflects a conflict rate between 4% and 5%. Figure 21(a) shows how the rate of out of order messages influences the number of transactions that need to be rescheduled and the overall response time of the system

As can be expected, the number of transactions that must be rescheduled grows with the rate of out of order messages. However, the rate is very small for all test runs. Even in the case in which the optimistic and total order are different for each individual pair of messages less than 1% of transactions must be rescheduled. The reason is that a transaction needs to be rescheduled only if it is wrongly ordered in regard to conflicting transactions. In non-conflicting cases the order does not matter. As a result, at low conflict rates, abort rates are always very small, even for the case that the spontaneous total order property does not hold.

The response times of transactions that need to be rescheduled is illustrated in Figure 21(b) and is roughly twice as long as the response time of transaction that do not need to be rescheduled. The relation between the response time of rescheduled and non-rescheduled transactions is related to the overlap between communication and transaction processing. If a transaction has to be rescheduled, it must be aborted, hence all the processing done before the `TO-delivery` is lost and has to be redone. If the time between `OPT-delivery` and `TO-delivery` is short, aborting and re-executing the wrongly executed operations is fast. If the overlap equals the time to process the transaction, then the response time is more than doubled for rescheduled transactions.

8 Conclusion

In this paper, we have presented a new way of integrating communication and database technology to build a distributed and replicated database architecture. Taking advantage of the characteristics of today's networks, we use an optimistic approach to overlap communication and transaction processing. In this way, the message overhead caused by the need for coordination among the sites of a distributed system is hidden by optimistically starting to execute transactions. Correctness is ensured by delaying transaction commitment until the message is definitively delivered.

The modularity of our approach is given by encapsulating message exchange using a group communication module, on which the transaction processing module bases the execution of transactions. The new solution can easily be integrated into existing systems because the modifications both on group communication and database side are straightforward and easy to implement. Our approach provides a solution whereby the benefits offered by group communication can be fully exploited.

The simulation results show that the approach can offer an interesting performance gain compared with existing solutions. We believe that it can be used in two different scenarios: firstly, on a local area network where the optimistic assumption holds most of the time. In this case, any type of database and workload will benefit from the approach and provide faster response times. Secondly, our approach will also work in wide area networks if the workload has low conflict rates. Here, even if the tentative order is completely different from the tentative order, abort rates are low. In this case, our algorithm basically reflects an optimistic protocol in which the total order provides the rule for which of two concurrent transactions has to be aborted in the case of conflicts.

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, Santa Barbara, California USA, 1996.

[2] G. Alonso. Partial database replication and group communication primitives (extended abstract). In *Proceedings of the* 2^{nd} *European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 171–176, Zinal (Valais, Switzerland), January 1997.

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

[4] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

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

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[7] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[8] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. Technical Report TR95-1527, Cornell University, Computer Science Department, 1995.

[9] R. Goldring. A discussion of database replication technology. *Info DB*, 1(8), May 1994.

[10] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996. ACM-SIGMOD.

[11] J. N. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[12] V. Hadzilacos and S. Toueg. *Distributed Systems, 2ed*, chapter 3, Fault-Tolerant Broadcasts and Related Problems. Addison Wesley, 1993.

[13] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[14] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333 – 379, 2000.

[15] B. Kemme, A. Bartoli, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks (DSN2001)*, Göteborg, Sweden, June 2001.

[16] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the International Conference on Distributed Computing Systems*, Austin, Texas, June 1999.

[17] Mesquite Software Inc., 3925 West Braker Lane Austin Texas TX-78759-5321, USA. *CSIM18 simulation engine (C++ version)*, 1994.

[18] 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.

[19] Concurrency control, transaction isolation and serializability in SQL92 and Oracle7. White paper, Oracle Corporation, Oracle Corporation, 500 Oracle Parkway, Redwoord City, CA 94065, 1995.

[20] F. Pedone. A closer look at optimistic replica control. In *Proceedings of the* 2^{nd} *European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 273–278, 1997.

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

[22] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the* 12^{th} *International Symposium on Distributed Computing (DISC'98, formerly WDAG)*, September 1998.

[23] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[24] D. Stacey. Replication: DB2, Oracle, or Sybase. *Database Programming & Design*, 7(12), 1994.

[25] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proceedings of the* 9^{th} *IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, October 2000.

[26] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15)*, Beppu City, Japan, February 2001.

[27] R. van Renesse, K. P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.