Using Optimistic Atomic Broadcast in Transaction Processing Systems *

Technical Report No. 325
ETH Zürich, Departement of Computer Science

Bettina Kemme* Fernando Pedone† Gustavo Alonso* André Schiper†

*Information and Communication Systems
Institute of Information Systems
Swiss Federal Institute of Technology (ETH)
ETH Zentrum, CH-8092 Zürich
E-mail: {kemme,alonso}@inf.ethz.ch
http://www.inf.ethz.ch/department/IS/iks/

†Operating Systems Laboratory
Computer Science Department
Swiss Federal Institute of Technology (EPFL)
IN-Ecublens, CH-1015 Lausanne
{Fernando.Pedone,Andre.Schiper}@epfl.ch
http://lsewww.epfl.ch/

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.

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 [CT91, BSS91] 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 [BC94, FvR95]. 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.

*Part of this work has been funded by Swiss Federal Institute of Technology (ETH and EPFL) within the DRAGON Research Project (Reg-Nr. 41-2642.5)

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 [PS98]. 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, realistic, and reasonable 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 [AAES96, KA98b, KA98a, PGS97, PGS98]. 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 further measures have to be taken to guarantee that the serialization order obeys the definitive total order. If the time it takes to receive confirmation of the order in which messages are delivered is comparable to the time it takes to execute a transaction, and there are not many cases in which the tentative order differs from the definitive order, then the overhead of the group communication mechanism has been hidden 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) 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 optimistic transaction processing is described in Section 4, and its correctness proof in Section 5. Queries are discussed in Section 6. In Section 7, we enhance our transaction model and allow fine granularity concurrency control. 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 exchanging messages. We assume asynchronous (no bound on the transmission delays) and reliable communication. Reliable communication ensures that a message sent by a site N_i to a site N_j is eventually received by N_j. 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 system using *Atomic Broadcast with Optimistic Delivery* and the transaction management providing *Optimistic Transaction Processing (OTP)*.

2.1 Atomic Broadcast with Optimistic Delivery

Communication 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.

Although there exist many different ways to implement total order delivery [CT91, BSS91, DM96, MMSA+96, vRBM96], 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 the messages arrive at all sites spontaneously totally ordered [PS98]. 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. Instead, one could 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 virtual time proposed by [Jef85].

In order to illustrate the spontaneous total order property of local area networks, we have conducted some experiments on a cluster of 4 sites (Ultrasparc 1 workstations) connected by an Ethernet network (10 Mbits/s), where all sites simultaneously send messages using IP multicast. Figure 1 shows the percentage of spontaneously ordered messages vs. the interval between two consecutive messages on each site. For example, for this configuration, if each site sends one message each 4 milliseconds, around 99% of the messages arrive at all sites in the same order.

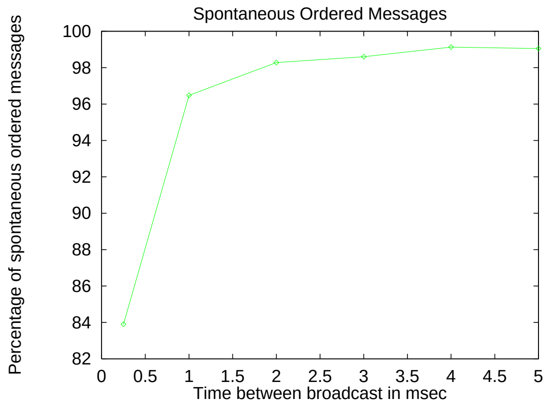


Figure 1: Spontaneous total order in a 4-site-system

The atomic broadcast with optimistic delivery used in this work, is formally defined by the three primitives shown below.

- `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 `OPT-delivering` m), but rather deliver only a confirmation message that contains the identifier of m. Further details and a complete specification of the atomic broadcast with optimistic delivery is given in Section 3.

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]

For simplicity, initially, we assume that all data access is done through stored procedures, with one transaction corresponding to one stored procedure. In the following, we use transaction and stored procedure as equivalent terms. Since the details of transactions are hidden by the use of stored procedures, we use a simplified version of the traditional transaction model [BHG87]. Transactions access the database by reading and writing (updating) objects of the database. Transactions execute atomically, i.e., a transaction T either commits or aborts all its results. It is possible for two or more transactions to access the database concurrently. Two transactions conflict if both access the same object X and at least one of the transactions updates X. A history H is a partial order of a set of transactions and reflects one possible execution. All conflicting transactions contained in H are ordered. 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 transactions and they order conflicting transactions in the same way. A history H is said to be serializable if it is conflict equivalent to some serial 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. In a replicated database system the combined concurrency control and replica control protocols provide 1-copy-serializability if the following holds:[2] $H = \bigcup_H H_i$ is serializable with $H_1, H_2, \ldots, H_n$ being the histories at sites $N_1, \ldots, N_n$.

In Section 7, we enhance the model to be able to serialize transactions on the level of a single operation and not on the entire transaction.

[1] In fact, many commercial databases base their replication solutions on the use of stored procedures [Sta94, Gol94]

[2] We define the union operation $\bigcup_H$ between histories as follows. Let $H = (\Sigma, <_H)$ be a history where Σ is a set of transactions, and $<_H$ is a set defining a transitive binary relation between transactions in Σ. If H' and H'' are two histories, then $H = H' \bigcup_H H''$ is such that $\Sigma = \Sigma' \bigcup \Sigma''$ and $<_H = <'_H \bigcup <''_H$.

2.3 Concurrency Control

For our purposes and to simplify the presentation, we assume a rather simple version of concurrency control. We assume that each stored procedure (or transaction) belongs to one of several disjoint *conflict classes*. The procedures of one conflict class only access a certain partition of the database, and different conflict classes work on different partitions. A conflict class is therefore determined by a set of objects, and transactions belonging to this class are only allowed to access these objects. This means that transactions within one conflict class have a high probability of having conflicts while it is assured that transactions from different conflict classes do not conflict.

With this assumption, concurrency control is done as follows (Figure 2). For each conflict class C there exists a *FIFO class queue* CQ. When a transaction T, $T \in C$, is started, it is added to CQ. When T is the only transaction in CQ, then it can be submitted to the data manager. When there are already other transactions queued in CQ, T has to wait. When a transaction commits (this only happens when the transaction is the first one in its class queue), it is removed from the queue and the next transaction waiting (if any) is submitted for execution. This means that whenever transactions are in the same conflict class, they are executed sequentially. When they are in different classes, their execution is not ordered. It is easy to see that this protocol guarantees serializability because conflicting transactions are fully serialized.

This mechanism is a simplified version of the standard lock table used in existing database systems [GR93]. The difference is that a lock table has a queue for each data item, and an entry corresponds to one transaction operation, i.e., a transaction can have several entries in a lock table, while our approach has a queue per disjoint class and a transaction is exactly queued in one class queue once when the transaction begins. Later on, we show how these same ideas can be applied to the more general case.

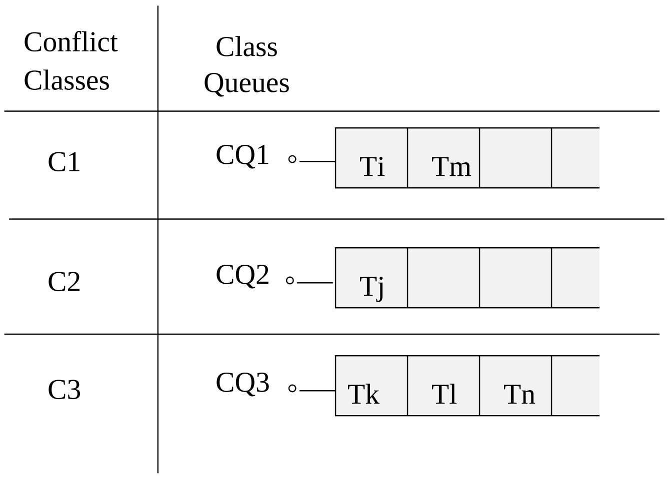


Figure 2: Conflict classes and their class queues

2.4 Execution Model

We use a variation of the read-one/write-all approach for replica control. When a user submits a query (i.e., a transaction that does not update any data) to a site N in the system, the query is executed locally at N. This is very important, because replication is mainly used to provide fast local read access [KA98b]. However, when a user sends the request for an update transactions to N, N `TO-broadcasts` the request to all sites so that the updates are executed at all sites. Stored procedures support this approach very well. Since they are predefined, the type of the transaction (query or update transaction) can be declared in advance. In the next sections, we first focus on update transactions only. Queries are considered in section 6.

Figure 3 depicts the coordination of the communication manager and the transaction manager to execute update transactions. The communication manager receives and orders `TO-broadcasted` requests. 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 corresponding transactions and adds them to the corresponding class queue. The *Execution* module executes the transactions of the class queues concurrently as long as they do not belong to the same conflict class. However, whenever they conflict, they are ordered according to the tentative order. If two transactions T_1 and T_2 conflict, and T_1 is tentatively ordered before T_2, then T_2 has to wait until T_1 commits before it can start executing. However, 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 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.

Atomic broadcast protocols have traditionally been defined by a single delivery primitive [CT91, BSS91, DM96, MMSA+96, vRBM96] that guarantees that no site delivers a message out of order (total order property [HT93]). Only recently, optimistic protocols that exploit the characteristics of the network, or the semantics of the application, have been considered. In [PS98], 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

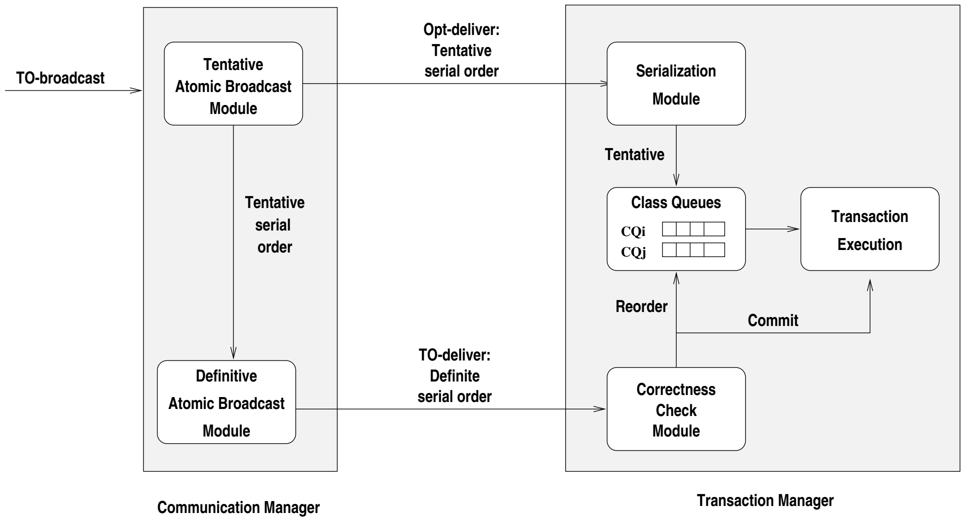


Figure 3: Execution model

protocol, there is a tradeoff between *optimistic* and *conservative* (non-optimistic) decisions. However, messages are never delivered in the wrong order to the application.

The approach proposed here is a more *aggressive* version of the protocol in [PS98], 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 belong to the same conflict class, total order between these messages is not necessary).

Different degrees of optimism are summarized in Figure 4 (for simplicity, we consider a scenario without failures). If messages are sent to all sites using network broadcast, there is a high probability that they will reach all sites in the same order.

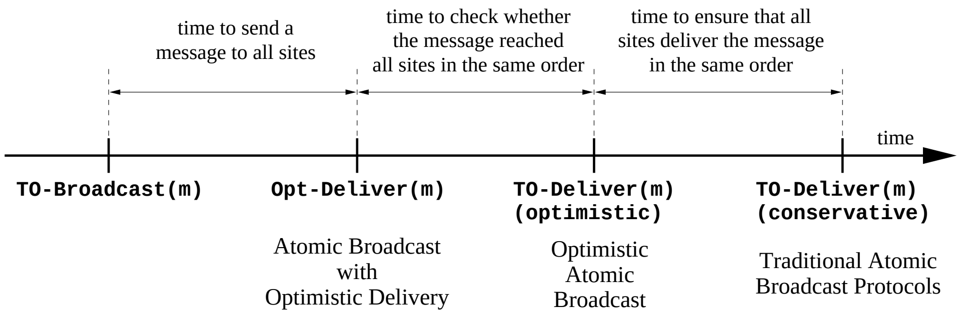


Figure 4: Degrees of optimism

The Atomic Broadcast with Optimistic Delivery (defined in Section 2.1) is specified by the following properties.

Termination: If a site `TO-broadcasts` m, then every site eventually `Opt-delivers` m and

`TO-delivers` m.

Global Agreement: If a site `Opt-delivers` m (`TO-delivers` m) then every site eventually `Opt-delivers` m (`TO-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 show how transactions are executed in the system, and present the OTP-algorithm for optimistic transaction processing.

4.1 General Idea

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

$$\textit{Tentative total order at } N: \; T_1,\ T_2,\ T_3,\ T_4,\ T_5,\ T_6$$

$$\textit{Tentative total order at } N': \; T_1,\ T_3,\ T_2,\ T_4,\ T_6,\ T_5$$

Assume as well that there are three different conflict classes and the distribution of the transactions is $T_1, T_2 \in C_x$, $T_3, T_4 \in C_y$, and $T_5, T_6 \in C_z$. When the scheduler receives the transactions in tentative order, it places them as follows in the queues:

$$\begin{array}{rrl} \textit{At } N: & CQ_x = & T_1,\ T_2 \\ & CQ_y = & T_3,\ T_4 \\ & CQ_z = & T_5,\ T_6 \\ \textit{At } N': & CQ_x = & T_1,\ T_2 \\ & CQ_y = & T_3,\ T_4 \\ & CQ_z = & T_6,\ T_5 \end{array}$$

The transaction manager will then submit the execution of the transactions at the head of each queue, i.e., T_1, T_3, and T_5 are executed at N, and T_1, T_3, and T_6 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:

$$Definitive\ total\ order: T_1,\ T_2,\ T_3,\ T_4,\ T_5,\ T_6$$

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_2 and T_3 and for transactions T_5 and T_6.

Upon receiving the messages in definitive total order, the transaction manager has to check whether what it did makes sense. At N, the tentative order and the definitive order are the same, thus, transactions T_1, T_3, and T_5 can be committed and the next transactions in the queues executed. Since the definitive ordering for T_2, T_4 and T_6 has also been established, once these transactions are completely executed they can also commit.

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_2 and T_3 is not really important because these two transactions do not conflict. However, the order between T_5 and T_6 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_6 and first perform the ones of T_5 before it reexecutes T_6.

It is trivial for the transaction manager of N' to detect such conflicts. Assume T_6 and T_5 have been `Opt-delivered` and T_6 is ordered before T_5 in the conflict queue CQ_z. When T_5 is `TO-delivered` (note, that T_5 is `TO-delivered` before T_6), the transaction manager of N' performs a correctness check. It looks in the queue and scans through the list of transactions. The first transaction is T_6 and T_6 has not yet been `TO-delivered`. The wrong order is detected and the updates of T_6 can be undone using traditional recovery techniques [BHG87]. T_6 will then be appended to the queue after T_5. 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_5 `TO-delivered` when it performs the check. When at a later timepoint T_6 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_5. Since T_5 is marked `TO-delivered` the transaction manager knows that this time the scheduling of T_5 before T_6 was correct and no rescheduling has to take place. Thus, T_6 is simply marked `TO-delivered`. Note, that the transaction manager does not need to realize that the tentative and the definitive order for T_2 and T_3 did not match.

With this method there is no need for the transaction manager to memorize the tentative and definitive orders. It only uses the primitives `Opt-deliver` and `TO-deliver` and marks `TO-delivered` transactions committable, because once `TO-delivery` has taken place, the transaction will not be aborted anymore. Whenever a transaction is marked committable and all its operations have been executed it can commit and be removed from the queue, so that the next transaction can be started. Reordering will take place only when transactions in the same queue are not in the right order, this means that when a transaction is `TO-delivered` but there are preceding transactions in the same queue which are not yet committable.

This example shows both the basic mechanisms used as well as the advantages of the approach. Namely, how communication and transaction execution can be done at the same time. Note also 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_2 and T_3 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 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 at most one transaction of each conflict class is executed at a 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 5, is to append `Opt-delivered` transactions to their corresponding conflict classes (S1), to mark that this serialization order is still tentative (S2), and to submit the execution of transactions when there are no conflicts (S4).

The execution module has to inform the transaction manager about completely executed transactions (Figure 6). 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 queue can be marked `executed`.

Upon `Opt-delivery` *of message m containing transaction T_i:*

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

Figure 5: Serialization Module

Upon complete execution of transaction T_i:

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

Figure 6: Execution Module

The correctness check module is activated upon `TO-delivery` of a transaction. Figure 7 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.

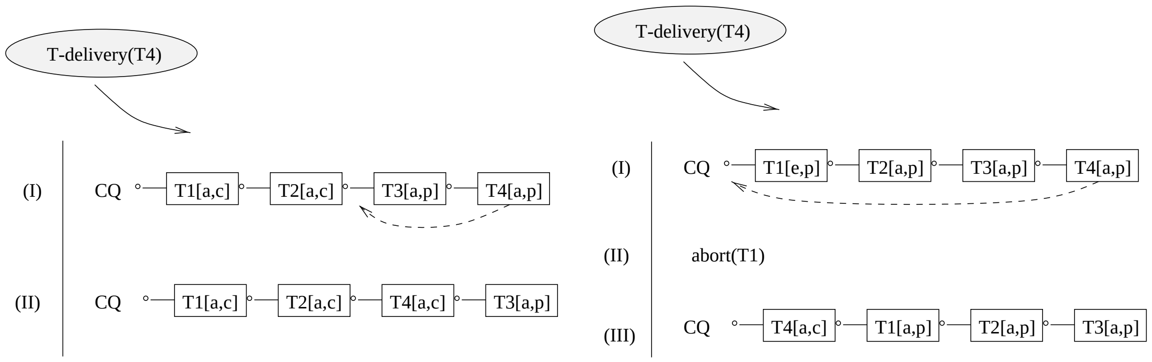
Since each message is `Opt-delivered` before it is `TO-delivered` (Local Order property), it is guaranteed that there is an entry for a transaction T in its corresponding class queue (CC1). The first transaction of a class queue commits whenever it is `TO-delivered` and totally executed (CC2,CC3) (both events must be true) and the execution of the next transaction in the class 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 reexecuted at a later point in time. The protocol guarantees that all `committable` transactions are ordered before all pending ones in the class queue CQ (due to step CC10). In particular, if transaction T of queue CQ is `TO-delivered` and the first transaction in CQ is still pending, all transactions before T are 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 8). 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 class queue has the following entries: $CQ = T_1[a,c],\ T_2[a,c],\ T_3[a,p],\ T_4[a,p]$. 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

Upon `TO-delivery` *of message m containing transaction T_i:*

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

Figure 7: Correctness Check Module



(a) (b)

Figure 8: Examples of reordering transactions: (a) without and (b) with abort

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 CQ has the entries: $CQ = T_1[e,p]$, $T_2[a,p]$, $T_3[a,p]$, $T_4[a,p]$. 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.

5 Correctness

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 belonging to conflict class C, 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 5.1 we assume a failure free execution.

Theorem 5.1 *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 class queue CQ.

1. *Induction Basis:* If T_i is the first transaction in CQ ($n = 1$), 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 CQ, 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 CQ. 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 CQ. When this happens, T_i is at most at position k, and hence, will eventually commit according to the induction hypothesis. □

Lemma 5.1 *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 belonging to the same conflict class C 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 CQ 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 CQ 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 5.2 *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. Lemma 5.1 proves that in all these histories conflicting transactions are always processed in the same order, namely the definitive order established by the atomic broadcast. Therefore, all local histories are conflict equivalent to each other. This guarantees the "1-copy" property, i.e., all the copies behave in the same way. Moreover, there is a serial history that is conflict equivalent to all those produced: the one derived from the definitive total order provided by the atomic broadcast (this provides the "serializability" property). □

The extension of the OTP-algorithm to include queries and fine granularity locking can be found in the next sections.

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 which can be processed locally while a certain amount of updates must be performed at all sites. 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 [BHG87, GR93, Ora95]. 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).

In what follows we sketch how two different solutions can be integrated into the replica control presented in this paper. The protocols are extensions of the OTP-algorithm for updating transactions described in Section 4.2.

The first solution takes into consideration that it is not reasonable to require that queries belong to a single conflict class. Since queries often access a lot of data, this approach would make it necessary to choose a very coarse granularity for the conflict classes, thereby reducing

Upon begin of query Q_i:

for each conflict queue CQ, Q_i wants to access
    Build an entry for Q_i and mark it `committable`
    if first transaction T_j in CQ is marked `pending`
        Abort T_j
    end if
    Insert the entry of Q_i before the first transaction T_k in CQ marked `pending`
    if Q_i is now the first transaction in CQ
        Submit the execution of Q_i
    end if
end for each

Figure 9: OTP-Q-protocol for queries

concurrency. Therefore, it is important to allow queries to belong to many conflict classes and hence, to spread their access across arbitrary partitions of the database.

Figure 9 depicts the OTP-Q-protocol. The start of a query is very similar to processing the `TO-delivery` of a transaction but with the difference that the query can access several conflict classes. For each conflict class, a query Q is scheduled after all committable and before all pending transactions. It is easy to see that the protocol provides serializability. The total order provided by the atomic broadcast is still an equivalent serial execution (i.e., whenever $T_i \rightarrow_{TO} T_j$ then the serialization order at all sites is $T_i \rightarrow T_j$). A query Q is included in this order as follows: Assume all transactions T_i are indexed according to the total order of the atomic broadcast, i.e., if T_i is `TO-delivered` before T_j, then $i < j$. Let i be the index of the last transaction whose `TO-delivery` was processed before Q starts. Q is added to the serialization order by simply ordering it directly after T_i and before T_{i+1} (we could imagine Q to have the index $i.5$).

Since queries are added to all their conflict classes at starting time, all data the transaction wants to access must be known in advance. Furthermore, queries can be very extensive, leading to considerable delay for succeeding update transactions. Hence, it may be desirable to handle queries dynamically, i.e., queries should neither need to know in advance which conflict classes they are going to access nor should transactions be delayed too long by queries.

However, the solution is not straightforward. Queries cannot simply be added to a class queue when they want to access an object of this class for the first time. Such a protocol would violate 1-copy-serializability. The problem is the fact that update transactions of different conflict classes could now be indirectly ordered by queries that access both classes at different times. For example, such a protocol would allow the following serialization orders for the queues CQ_x and CQ_y at sites N and N':

$$\begin{array}{rrl} At\ \ N: & CQ_x = & T_1,\ T_2,\ Q,\ T_3 \\ & CQ_y = & T_4,\ Q,\ T_5,\ T_6 \end{array}$$

$$
\begin{aligned}
At\ N':\ \ CQ_x = &\quad T_1,\ Q',\ T_2,\ T_3 \\
CQ_y = &\quad T_4,\ T_5,\ Q',\ T_6
\end{aligned}
$$

This means that Q implicitly builds the serialization order $T_2 \rightarrow Q \rightarrow T_5$ at site N, while Q' leads to the order $T_5 \rightarrow Q' \rightarrow T_2$ at N' [Alo97].

To avoid such situation, we have to ensure that the execution at all sites is equivalent to the order induced by the atomic broadcast (as it is the case in the previous algorithm). To combine 1-copy-serializability with dynamic queries and fast execution for updating transactions, our last proposal uses snapshots for queries (similar to Oracle snapshots [Ora95]). To provide consistent snapshots for queries, different versions of the data of a conflict class are maintained, each labeled with the index of the transaction that created the version (again assuming that transactions are indexed according to their `TO-delivery`). A query receives an index when it starts. As in the previous algorithm, if T_i was the last processed `TO-delivered` message, the index for the query is $i.5$. When a query $Q_{i.5}$ wants to access a conflict class C for the first time, it receives a snapshot of the data that has been created by transaction T_j, where $j = max(k), k \leq i, T_k \in C$. With this, we produce the same execution order as with the OTP-Q-protocol. Note that this approach is similar to the hybrid algorithm in [KA98b].

7 Fine Granularity Locking

Class queues as a concurrency model have allowed us to demonstrate in a simple manner the idea of optimistic transaction processing. However, requiring that transactions may only access objects of a certain partition is quite restrictive. In some applications this might result in very few classes and hence in a very low degree of concurrency. This section extends the model to show how it can be applied in more generic settings: we drop the requirement that an update transaction may only access objects of a single conflict class. Instead 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 sites executes the corresponding stored procedure. But instead of including the request into a single class queue (corresponding to a partition), we now use a traditional lock table. The lock table maintains lock queues for each object of the database and transactions place lock entries into the queues of those objects they want to access. In our approach, placing the lock entries of a transaction is done in an atomic step at the time the transaction is delivered. This means that there is no interleaving with any other transactions. In other words, the important step for our approach is not how many queues there are but that transactions must be placed in all the necessary queues one at a time. Note that we require that all locks of a transaction are known in advance. We are aware that this is still restrictive since it relies on predefined stored procedures where one can tell in advance which fine-granularity objects are accessed by the transaction. We are working on improving our concurrency model so that it also allows other transaction types, e.g., interactive transactions.

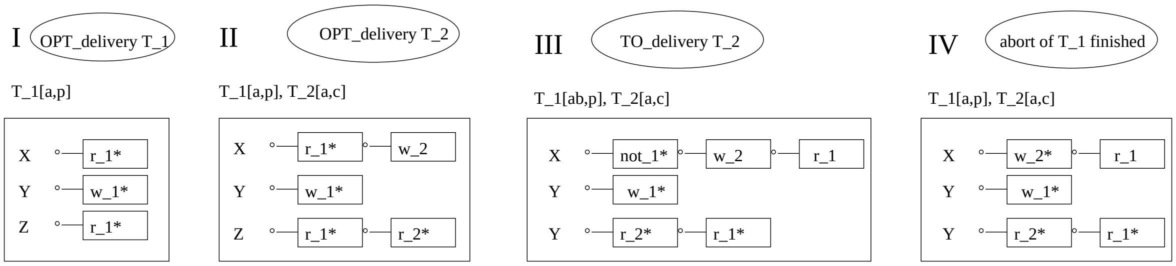


Figure 10: Example of fine granularity locking

7.1 Enhanced Transaction Model

In this section we enhance our original transaction model to fit the traditional transaction model as presented in [BHG87]. We now characterize a transaction (stored procedure) as a sequence of read $r_i(X)$ and write $w_i(X)$ operations on objects X. As before, update transactions 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 $o_i(X_1), \ldots, o_i(X_n)$. Conflicts are no more defined between transactions but rather between operations. 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. As described in section 2.2, a global history is the union of all local histories in the system. A history is now serializable if it orders all conflicting operations in the same way as a serial history.

Concurrency control is done via locking. 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 included in first-in-first-out 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. This is necessary because we have to include the lock entries of a transaction in an atomic step. This atomicity can be accomplished, e.g., by acquiring a latch on the lock table during lock insertion.

7.2 Example

As before, a transaction is sent to all sites. The main idea is for each transaction to request all its read and write locks when it is `Opt-delivered`. Once all lock requests are included in the lock table, the transaction starts executing. As before, once a message is `TO-delivered` the serialization order has to be checked.

Figure 10 depicts an example which shows the main difference between the enhanced system and the class queue 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 (V) 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. Like in the case of class queues, the correctness check module scans through all of T_2's locks and looks whether there exist locks of `pending` transactions that are ordered before the waiting locks of T_2. Note that all locks of a single transactions 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. 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' 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. It 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.

7.3 Algorithm

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

Figure 11 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`. Only after the locks are requested the transaction starts executing (S5).

	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 11: Fine granularity serialization module

The execution module (Figure 12) 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` (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. 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 13) 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 (CC13-CC14).

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:
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 12: Fine Granularity Execution Module

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 (CC15-CC19). 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.

Figure 14 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`)`, `OPT-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). Its state is `active` and `pending`. When T_2 is `OPT-delivered` (step II) its lock on Z

```
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         Schedule T_i's lock before the first lock entry
CC14         that belongs to a pending transaction T_k.
CC15         for each conflicting granted lock of a pending transaction T_j
CC16             Keep a notification entry at the begin of the queue.
CC17             Only when this entry is released (see execution module E16-E18),
CC18             T_i's lock can be granted
CC19         end for each
CC20     end for each
CC21 end if
```

Figure 13: Fine Granularity Correctness Check Module

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-CC12). T_2's locks on X and Z are scheduled before T_1's locks (CC13). 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 (CC15-CC19). 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 reorder 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 (CC13-CC14). 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 (CC15-CC19). 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

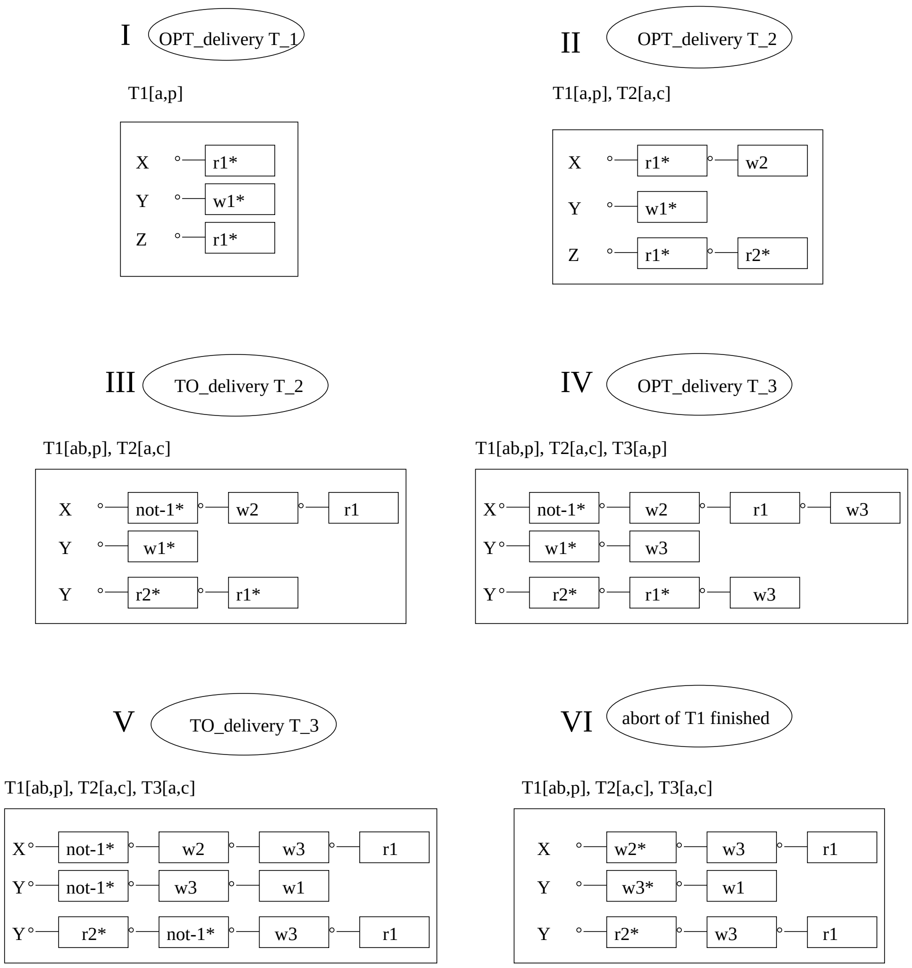


Figure 14: Reordering with fine granularity locking

be committed.

Note that execution state and delivery state are orthogonal to each other. Figure 15 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.

7.4 Proof of Correctness

The proof of correctness follows the same line as the one provided in section 5. We first want to show that 1-copy-serializability is provided. In the simple OTP-algorithm the definitive total order of the atomic broadcast ordered entire transactions. Now, it only orders conflicting operations.

Lemma 7.1 *Using the FG-algorithm, each site orders and executes conflicting operations in the definitive order provided by the atomic broadcast.*

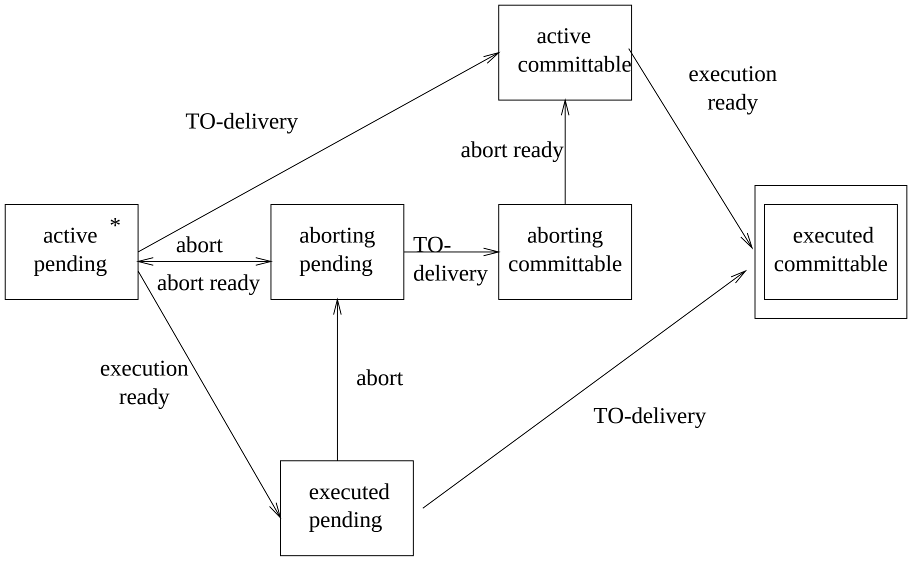


Figure 15: Execution and delivery states of transactions

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 5. 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 reexecute the operation when T_i has committed. □

Theorem 7.1 *The FG-algorithm provides 1-copy-serializability.*

Proof

Again the proof is very similar to the one provided in section 5. All local histories contain exactly the same update transactions. Since all conflicting operations are executed in the same order at all sites, namely the definitive order of the atomic broadcast, the "1-copy" property is guaranteed. The serial history that is conflict equivalent to all those produced is the one derived from the definitive total order established by the atomic broadcast ("serializability" property). □

Finally we want to prove that the protocol is starvation free.

Theorem 7.2 *The FG-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 (CC13-CC14). 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.

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 group communication semantics, 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 without loss of performance.

References

[AAES96] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. Technical report, Department of Computer Science, Santa Barbara, 1996.

[Alo97] G. Alonso. Partial database replication and group communication primitives. In *2nd Europ. Research Seminar on Advances in Distr. Systems (ERSADS'97)*, Zinal (Switzerland), March 1997.

[BC94] K. Birman and T. Clark. Performance of the Isis distributed computing toolkit. Technical report, Departement of Computer Science, Cornell University TR-94-1432, June 1994.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Massachusetts, 1987.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CT91] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proc. of the 10th ACM Symp. on Principles of Distributed Computing*, pages 325–340, August 1991.

[DM96] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):63–70, April 1996.

[FvR95] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. Technical report, Departement of Computer Science, Cornell University TR-95-1527, July 1995.

[Gol94] R. Goldring. A discussion of relational database replication technology. *InfoDB*, 8(1), 1994.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[HT93] V. Hadzilacos and S. Toueg. *Distributed Systems, 2ed*, chapter 3, Fault-Tolerant Broadcasts and Related Problems. Addison Wesley, 1993. Edited by S. Mullender.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[KA98a] B. Kemme and G. Alonso. Database replication based on group communication. Technical report, Department of Computer Science, ETH Zürich, No. 289, February 1998.

[KA98b] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the Int. Conf. on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

[MMSA+96] 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.

[Ora95] Oracle. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*, 1995. White Paper.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *16th IEEE Symp. on Reliable Distributed Systems (SRDS'97)*, Durham, USA, October 1997.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of EuroPar*, Southampton (England), September 1998.

[PS98] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proc. of the 12th Int. Symp. on Distributed Computing (DISC'98)*, September 1998.

[Sta94] D. Stacey. Replication: DB2, Oracle, or Sybase. *Database Programming & Design*, 7(12), 1994.

[vRBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Comm. of the ACM*, 39(4):76–83, April 1996.