Processing Transactions over Optimistic Atomic Broadcast Protocols *

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 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 which, based on optimistic assumptions about the communication system, reduces the average delay for message delivery. In this paper, we develop this idea further and present a replicated database architecture that employs the new atomic broadcast primitive in such a way that the coordination phase of the atomic broadcast is fully overlapped with the execution of transactions, providing high performance without relaxing transaction correctness.

1. Introduction and Motivation

Atomic Broadcast [6, 5] primitives are a well known mechanism to increase fault tolerance and to provide a semantically rich framework to develop distributed systems. Unfortunately, it is also recognized that atomic broadcast suffers from scalability problems [4, 8] as it involves coordination between sites before messages can be delivered.

Recently, a new atomic broadcast protocol has been proposed which, based on optimistic assumptions about the network, reduces the average delay for message delivery to the application [18]. This protocol uses the order in which messages arrive at each site as a first optimistic guess, and only if a mismatch of messages is detected, further coordination between the sites is needed to agree on a total order.

*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)

In this paper we develop this idea further, and show how applications can take advantage of the optimistic assumption by overlapping the coordination phase of the atomic broadcast algorithm with the processing of transactions. 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, 12, 11, 16, 17, 19]. 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. The transaction manager uses this tentative total order to determine a scheduling order for the transaction and starts executing the transaction. The commitment of the transaction, however, is postponed. 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 message order is comparable to the time it takes to execute a transaction, and tentative and definite order are mostly the same, then the overhead of atomic broadcast mechanism has been hidden behind the cost of executing a transaction.

The results reported in this paper make two important contributions. Firstly, our solution avoids most of the over head of atomic broadcast by overlapping the processing of messages (execution of transactions) with the algorithm used to totally order them. Secondly, our approach compares favorably with existing commercial solutions for database replication [20] in terms of performance and consistency. While most systems achieve performance by using asynchronous replication mechanisms (update coordination is done after transaction commit) our solution offers comparable performance and at the same time maintains global consistency.

The paper is structured as follows. In Section 2 we describe the system model. In Section 2.1 we present the atomic broadcast primitive used in our database algorithms. The optimistic transaction processing is described in Section 3, and its correctness proof in Section 4. Queries are discussed in Section 5, and Section 6 concludes the paper.

2. Model and Definitions

A replicated database consists of a group of sites $N = \{N_1, N_2, \ldots, N_n\}$ which communicate by exchanging messages. We assume asynchronous (no bound on transmission delays) and reliable communication (a message sent by N_i to 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. Data is accessed by executing transactions.

2.1. Atomic Broadcast with Optimistic Delivery

Communication is based on *atomic broadcast* providing an ordering of all messages in the system, i.e., all sites receive all messages in the same order.

Although there exist many different approaches on how to implement total order [6, 5, 7, 14, 21], 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 [18]. To illustrate the spontaneous total order property of local area networks, Figure 1 describes an experiment conducted 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. The figure 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.

Only recently, an Optimistic Atomic broadcast protocol that takes advantage of these characteristics has been proposed [18]. It first checks whether the order in which messages are received is the same at all sites. If so, the algorithm does not require any further coordination between sites to reach an agreement on the order of such messages. Since the verification phase introduces some additional messages, there is a tradeoff between *optimistic* and *conservative* (non-optimistic) decisions. However, messages are never delivered in the wrong order to the application.

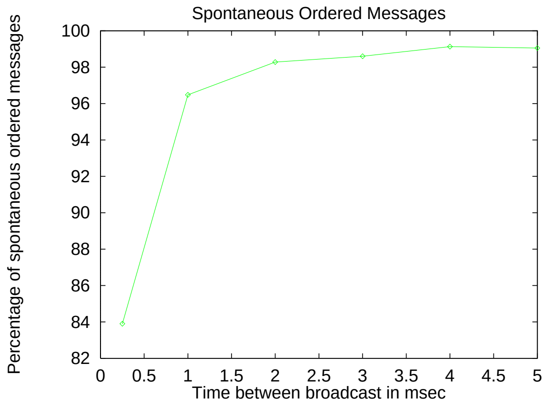

Figure 1. Spontaneous total order in a 4-site-system

The approach proposed here is a more *aggressive* version of the protocol in [18]. If a spontaneous total order is the normal case, it seems to be a waste of time to delay the delivery of a message until the sites agree to this same total order. Instead, one could optimistically process messages as they arrive. If a message is processed out of order, one has to pay the penalty of undoing what has been processed so far, and redo it again in the proper order. This approach is conceptually similar to virtual time [10].

We define an atomic broadcast with optimistic delivery by three primitives. `TO-broadcast(`m`)` broadcasts the message m to all sites in the system. `Opt-deliver(`m`)` delivers a message m optimistically to the application once it is received from the network. `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* and is the same at all sites. In practice, `TO-deliver(`m`)` will not deliver the entire body of the message (which `OPT-deliver(`m`)` already did), but rather deliver only a confirmation message that contains the identifier of m.

The Atomic Broadcast with Optimistic Delivery 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.

2.2. Transaction Model

One way to interact with a database 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. Since 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 the purposes of this paper, we assume that all data access is done through stored procedures, with one transaction corresponding to one stored procedure. Since the details of transactions are hidden by the use of stored procedures, we use a simplified version of the traditional transaction model [3]. 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 committed transactions and reflects one possible execution. All conflicting transactions contained in H are ordered. A history H is serial if it totally orders all transactions. 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.

The correctness criterion for replicated database systems is *1-copy-serializability*: despite the existence of multiple copies an object appears as one logical copy and the system only allows serializable histories. Formally, a database system provides 1-copy-serializability if for all possible executions the following holds:[2] $H = \bigcup_H H_i$ is serializable with $H_1, H_2, \ldots, H_n$ being the histories at sites $N_1, \ldots, N_n$.

[1] In fact, many commercial databases base their replication solutions on the use of stored procedures [20].

[2] 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 $<_H = <'_H \bigcup <''_H$.

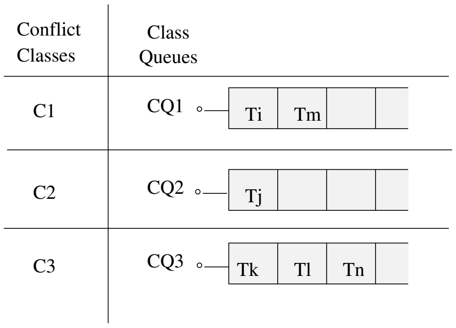

Figure 2. Conflict classes and their class queues

2.3. Concurrency Control

Serializing the execution of conflicting transactions is achieved through concurrency control. To simplify the presentation, we assume a rather simple model of concurrency control. We assume that each stored procedure (or transaction) belongs to one of several disjoint *conflict classes*. The transactions of one class are only allowed to access objects of a certain partition of the database and different conflict classes work on different partitions. This means that transactions within one conflict class have a high probability of having conflicts while 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 $T \in C$, is started, it is added to CQ. When T is the only transaction in C then its execution can be started. When there are already other transactions queued in C, T has to wait. When a transaction commits (it must be the first one in its queue), it is removed from the queue and the next transaction waiting starts to execute. Transactions in the same conflict class 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 [9]. The differences are that a transaction may enter step by step several entries into a lock table, one for each data it accesses (fine granularity 2-phase locking), while our approach queues a transaction in exactly one class queue when the transaction begins. [3]

[3] This approach has the advantage that transactions are never involved in deadlocks, which is the case with lock tables.

We would like to note, however, that the ideas of this paper can also be applied to a more general concurrency control model. In [13], we present solutions using finer granularity locking.

2.4. Replica Control

We use a variation of the read-one/write-all (available) approach for *replica control* [3]. When a user submits a query (i.e., a transaction that does not update any data) to site N, the query is executed locally at N. However, when a user sends the request for an update transaction to N, N 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. We first focus on update transactions only. Queries are considered in Section 5.

The basic idea to guarantee the same serialization order at all sites is to use the total order provided by the atomic broadcast to decide on the order of conflicting transactions. This means that whenever two transactions belong to the same conflict class we guarantee that they are executed and committed in the order they are `TO-delivered`.

3. Optimistic Transaction Processing

In this section, we show how transactions are executed in the system, and present the OTP-algorithm for optimistic transaction processing.

3.1. Execution Model

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. 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, i.e. it determines the serialization order on behalf of the `Opt-delivered` messages. 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.

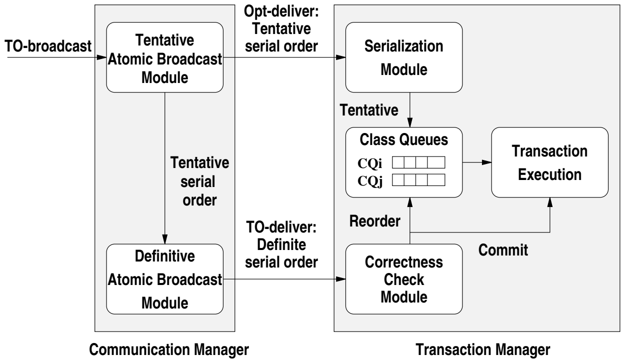

Figure 3. Execution model

3.2. General Idea of the Algorithm

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.

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

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:

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

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'. The transaction manager will wait to commit a transaction until it is totally executed and its 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 T_2 and T_3 and for 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.

At N', the total order is not the same as the tentative order. However, the ordering between T_2 and T_3 is not relevant because these two transactions do not conflict. Hence, the transaction manager does not need to realize the mismatch. 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.

To detect such conflicts the transaction manager marks transactions when they are `TO-delivered`. Assume the point in time when 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 is not marked `TO-delivered`. The wrong order is detected and the updates of T_6 can be undone using traditional recovery techniques [3]. T_6 will then be appended to the queue after T_5. The transaction manager then marks T_5 `TO-delivered`. 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, whenever transactions do not conflict, the discrepancy between the tentative and the definitive orders does not lead to any overhead (see T_2 and T_3 at N'). Hence, 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.

3.3. Algorithm

In the following, we present the OTP-algorithm for optimistic transaction processing. For simplicity, we divide the algorithm into different parts according to the different modules described in Section 3.1. 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 access control between the modules is necessary (for instance, by using semaphores).

Upon `Opt-delivery` *of message m containing transaction T_i:*
S1 Append T_i to the corresponding 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 4. Serialization Module

Upon complete execution of transaction T_i of class C:
E1 **if** T_i is marked `committable` (see next module)
E2 Commit T_i and remove T_i from CQ
E3 Start executing the next transaction in CQ
E4 **else**
E5 Mark T_i `executed`
E6 **end if**

Figure 5. Execution Module

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 (Figure 4). It appends `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 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 queue can be marked `executed`.

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

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

CC1 Look for the entry of T_i in CQ
CC2 **if** T_i is marked executed
(can only be the first txn in CQ)
CC3 Commit T_i and remove it from CQ
CC4 Start executing the next txn in CQ
CC5 **else** (not fully executed or not the first txn)
CC6 Mark T_i committable
CC7 **if** first txn T_j in CQ is marked pending
CC8 Abort T_j
CC9 **end if**
CC10 Schedule T_i before first txn T_k in CQ
that is marked pending
CC11 **if** T_i is now the first txn in CQ
CC12 Submit the execution of T_i
CC13 **end if**
CC14 **end if**

Figure 6. Correctness Check Module

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. In the following we use the shortcuts a for active, e for executed, p for pending and c for committable. Firstly, a class queue has the following entries

$CQ = \ T_1[a,c],\ T_2[a,p],\ T_3[a,p]$.

This means that T_1 has been TO-delivered, but not T_2 and T_3 and the execution of T_1 is still in progress. If now T_3 is the next one to be TO-delivered (before T_2) it is simply rescheduled between T_1 and T_2 (CC10). Since, the first transaction, T_1, is committable (it only waits for its execution to finish) it will not be aborted. Hence the result is

$CQ = \ T_1[a,c],\ T_3[a,c],\ T_2[a,p]$.

In the second example, the queue CQ has the form:

$CQ = \ T_1[e,p],\ T_2[a,p],\ T_3[a,p]$.

This means that none of the transactions is TO-delivered but T_1 is already executed. In this case, if T_3 is the first one to be TO-delivered (before T_1 and T_2), the first transaction T_1 must be aborted since it is still pending (CC8). After this, T_3 can be rescheduled before T_1 and submitted. This means that the execution of T_1 is rescheduled after the execution of T_3 and the result is

$CQ = \ T_3[a,c],\ T_1[a,p],\ T_2[a,p]$.

4. Correctness

Note that in this section we only look at update transactions. Queries are handled in the next section. In what follows 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, 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 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 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 at positions $n \leq k$, for some $k \geq 1$, 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 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 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 4.2 *The OTP-algorithm provides 1-copy-serializability.*

Proof Since all sites execute the same update transactions, the local histories of all sites contain 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 provided 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 established by the atomic broadcast ("serializability" property). □

5. 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 system where the main load are read-only 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 queries.

In particular, it is not reasonable to require queries to belong to a single conflict class but allow the access to multiple classes. Since queries often access a lot of data, access to a single class would make it necessary to choose a very coarse granularity for the conflict classes, thereby reducing concurrency. Furthermore, queries should be handled dynamically, i.e., queries neither need to know in advance their conflict classes nor should update transactions be delayed too long by preceding queries (what would be the case if queries are added to their classes when they start).

However, 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. For example, such a protocol would allow the following serialization orders for the queues CQ_x and CQ_y at sites N and N':

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

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' [2].

To avoid such situation, we ensure that the execution at all sites is equivalent to the total order of the atomic broadcast (i.e., whenever $T_i \rightarrow_{TO} T_j$ then the serialization order at all sites is $T_i \rightarrow T_j$). To combine 1-copy-serializability with dynamic queries and fast execution for updating transactions, we use snapshots for queries (similar to Oracle snapshots [15]). To provide consistent snapshots for queries, different versions of the data of a conflict class are maintained. Each data is labeled with the index of the transaction that created the version (assuming that transactions are indexed according to their `TO-delivery`, i.e. if T_i is `TO-delivered` before T_j, the $i < j$). A query receives an index when it starts. 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 a serialization order that obeys the total order.

6. Conclusion

In this paper, we present 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.

We are aware that our concurrency model is restrictive in that defining conflict classes and using stored procedures is only feasible for applications in which coarse-granularity locking does not result in performance degradation, or where one can tell in advance which fine-granularity objects are accessed by all transactions. We are working on improving our concurrency model so that it accepts the same type of operations as in traditional systems.

Acknowledgments

We want to thank the anonymous reviewers for their helpful advice.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Euro-Par'97*, Passau (Germany), August 1997.

[2] 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.

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

[4] 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.

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

[6] 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.

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

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

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

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

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

[12] 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.

[13] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Using optimistic atomic broadcast in transaction processing systems. Technical report, Department of Computer Science, ETH Zürich, March 1999.

[14] 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.

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

[16] 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.

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

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

[19] M. Raynal. Consensus-based management of distributed and replicated data. *Bulletin of the Techn. Comm. on Data Engineering*, 21(4), 1998.

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

[21] 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.

[22] A. Wool. Quorum systems in replicated databases: Science or fiction? *Bulletin of the Techn. Committee on Data Engineering*, 21(4), 1998.