Scalable Replication in Database Clusters

M. Patiño-Martínez[1], R. Jiménez-Peris[1], B. Kemme[2], and G. Alonso[2]

[1] Technical University of Madrid, Facultad de Informática,
Boadilla del Monte, Madrid, 28660, Spain,
{mpatino,rjimenez}@fi.upm.es,
http://lml.ls.fi.upm.es/mpatino, http://lml.ls.fi.upm.es/rjimenez

[2] Swiss Federal Institute of Technology (ETHZ), Institute of Information Systems,
ETH Zentrum, CH-8092, Zürich, Switzerland
{kemme,alonso}@inf.ethz.ch,
http://www.inf.ethz.ch/department/IS/iks

Abstract. The widespread use of clusters and web farms has increased the importance of data replication. In existing protocols, typical distributed system solutions emphasize fault tolerance at the price of performance while database solutions emphasize performance at the price of consistency. In this paper, we explore the use of data replication in a cluster configuration with the objective of providing both fault tolerance and good performance without compromising consistency. We do this by combining transactional concurrency control with group communication primitives. In our approach, transactions are executed at only one site so that not all nodes incur in the overhead of parsing, optimizing, and producing results. To further reduce latency, we use an optimistic multicast approach that overlaps transaction execution with the total order message delivery. The techniques we present in the paper provide correct executions while minimizing overhead and providing higher scalability.

1 Introduction

Data replication is considered a proven technique to enhance the fault-tolerance and performance of distributed applications. In practice, however, there is a wide gap between theory and practice. Conventional algorithms emphasize fault tolerance and use replication to implement fail over mechanisms [BHG87]. Database designers purposefully ignore these algorithms due to their poor performance [GHOS96]. Instead, most database products use *lazy* replication that neither provides fault-tolerance nor full consistency [GHOS96].

It has been suggested [KA98,KA,PGS98,AAAS97] that this gap could be bridged by combining database replication with group communication primitives [BR93] (mainly total order broadcast [HT93]). This line of work has resulted in efficient *eager* replication protocols that guarantee consistency and increase fault tolerance. Such results are especially suitable for clusters of computers and large collections of shared nothing databases. Although some initial optimizations have been suggested [KPAS99a] based on optimistic techniques [PS98], existing protocols have still two major drawbacks. One is the amount of redundant work performed at all sites. The other is the high abort rates created when consistency is enforced.

In this paper, we address these two issues. First, we present a protocol that minimizes the amount of redundant work in the system. Transactions, even those over replicated data, are executed at only one site. The other sites in the system only need to install the final changes. With this, and unlike in many other data replication protocols, the aggregated computing power of the system actually increases as more nodes are added. This is a great advantage in environments where transaction processing represents a significant overhead. For instance, in a typical web-farm, a transaction is written in SQL and results are returned in the form of web pages. Processing the transaction involves parsing the SQL, actually executing the transaction, generating the web pages and delivering them to the client. Obviously, if this is done at all sites, the amount of wasted resources can be very high. Moreover, the protocol exploits an extreme form of optimistic broadcast that hides most of the communication overhead behind the transaction execution. The only negative aspect of this protocol is that, in some situations, it aborts transactions in order to guarantee serializability.

To reduce the amount of aborted transactions, we propose a second algorithm. This second algorithm uses a transaction reordering technique that avoids aborts even when the optimistic and the total message orderings are not the same. Consistency is still guaranteed without higher transaction latency and the overall throughput considerably increases by decreasing the abort rate.

The paper is organized as follows. In Section 2 the system model and some definitions are introduced. Sections 3 and 4 describe the algorithms. Fault tolerance aspects of the algorithms are discussed in Section 5. Section 6 presents correctness proofs. Section 7 concludes the paper.

2 System Model

A replicated database consists of a group of nodes $N = \{N_1, N_2, ..., N_n\}$, also called sites, which communicate by exchanging messages. Sites only fail by crashing (byzantine failures are excluded) and we assume there is always at least one available node in the system. Each site contains a copy of the entire database.

2.1 Communication Model

Sites communicate using group communication primitives [BR93]. These primitives can be classified attending to the order guarantees and fault-tolerance provided [HT93]. *FIFO ordering* delivers all messages sent by a site in FIFO order. *Total order* ensures that messages are delivered in the same order at all the sites. In regard to fault-tolerance, *reliable multicast* ensures that a message is delivered at all available sites. *Uniform reliable multicast* ensures that a message that is delivered at a site (even if it is faulty) will be delivered at all available sites. We assume a virtual synchronous system [BR93], where all group members perceive membership (view) changes at the same virtual time, i.e., two sites deliver exactly the same messages before installing a new view.

In this paper we use an aggressive version [KPAS99b] of the optimistic total order broadcast presented in [PS98]. Each message corresponds to a transaction. Messages are optimistically delivered as soon as they are received and before the definitive ordering is established. With this the execution of a transaction can overlap with the calculation of the total order. If the initial order is the same as the definitive order, the transactions can simply be committed. If the final order is different, additional actions have to be taken to guarantee consistency. This optimistic broadcast is defined by three primitives [KPAS99b]. *To-broadcast*(m) broadcast the message m to all the sites in the system. *Opt-deliver*(m) delivers message m optimistically to the application (with no order guarantees). *To-deliver*(m) delivers m definitively to the application (in a total order). This means, messages can be opt-delivered in a different order at each site, but are to-delivered in the same total order at all sites. A sequence of opt-delivered messages is a *tentative order*. A sequence of to-delivered messages is the *definitive order* or total order. Furthermore, this optimistic multicast primitive ensures that every to-broadcast message is eventually opt-delivered and to-delivered by every site in the system. It also ensures that no site to-delivers a message before opt-delivering it.

2.2 Transaction Model

Clients interact with the database by issuing transactions. Transactions are partially ordered sets of read (r) and write (w) operations. Two transactions conflict, if they access the same data item and at least one of them is a write operation. A history H of committed transactions is serial if it totally orders all the transactions. Two histories H_1 and H_2 are conflict equivalent, if they are over the same set of transactions and order conflicting operations in the same way. A history H is serializable, if it is conflict equivalent to some serial history [BHG87]. For replicated databases, the correctness criterion is one-copy-serializability [BHG87]. Using this criterion, each copy must appear as a single logical copy and the execution of concurrent transactions must be equivalent to a serial execution over all the physical copies.

In this paper, concurrency control is based on *conflict classes* [KPAS99a]. Each conflict class represents a partition of the data. Transactions accessing the same conflict class have a high probability of conflicts, as they can access the same data, while transactions in different partitions do not conflict and can be executed concurrently. In [KPAS99a] each transaction must access a single *basic* conflict class (e.g., C_x). We generalize this model and allow transactions to access *compound conflict classes*. A compound conflict class is a non-empty set of basic conflict classes (e.g., $\{C_x, C_y\}$). We assume that the (compound) conflict class of a transaction is known in advance.

Each site has a queue CQ_x associated to each basic conflict class C_x. When a transaction is delivered to a site, it is added to the queues of the basic conflict classes it accesses. This concurrency control mechanism is a simplified version of the lock table used in databases [GR93]. In a lock table there is a queue for each data item, whilst in our approach each queue corresponds to an arbitrary set of data items (i.e., a conflict class).

Although the model may seem restrictive, it has real applications and it is used in practice. For instance, in e-commerce applications (commonly implemented as web farms), transactions do not randomly access all the database. Instead, they typically access only one or two lines of products. By partitioning the data and allocating partitions to different nodes, performance can be significantly improved. Our protocols take advantage of this fact to minimize the overall overhead.

2.3 Execution Model

Each conflict class (unitary or not) has a *master* site. We use a *read-one/write-all available* approach. Queries (read only transactions) can be executed at any site using a snapshot of the data (i.e., they do not interfere with update transactions). Update transactions are broadcast to all sites, however they are only executed at the master site of their conflict class. We say a transaction is *local* to the master site of its conflict class and is *remote* to the rest of the sites.

For instance, assume two sites N and N', where N is the master of conflict class $\{C_x\}$ and N' is the master of conflict class $\{C_x, C_y\}$. Then, a transaction only accessing C_x will be executed at N but not at N'. A transaction accessing both C_x and C_y will be executed at N' but not at N. Conflict classes are statically assigned to sites, but in case of failures, they are reassigned to different sites.

3 Increasing Scalability

This algorithm extends the one described in [KPAS99b] for fine-granularity locking by executing transactions at only one site and allowing transactions to access more than one conflict class. With these characteristics, this algorithm greatly improves scalability as the processing capability of the system increases as more sites are added. We call this algorithm NODO (NOn-Disjoint conflict classes and Optimistic multicast).

3.1 The Problem

When considering the scalability of data replication protocols, it is important to keep in mind that replication, by its very nature, does not always scale if the update ratio is high. To illustrate this point, consider a centralized system, which is capable of processing t transactions per second. Now assume a system with n nodes, all of them identical to the centralized one. Assume that the fraction of updates is w. Assume the load of local transactions at a node is x transactions per second. Since nodes must also process the updates that come from other nodes, the following must hold: $x + w\,(n-1)\,x = t$, that is, a node processes x local transactions per second, plus the percentage of updates arriving at other nodes that access replicated data $(w\,x)$ times the number of nodes. From here, the number of transactions that can be processed at each node is given by:

$$\frac{t}{1 + w\,(n-1)}$$

The total capacity of the system is n times that expression which yields, with t normalized to 1:

$$\frac{n}{1 + w\,(n-1)}$$

This expression has a maximum of n when $w = 0$ (there are no updates) and a minimum of 1 when $w = 1$ (all operations are updates).

Thus, in any replicated application, as the *update factor* w approaches 1, the total capacity of the system tends to that of a single node, independently of how many nodes are in the system. Note that the drop in system capacity is very sharp. For 50 nodes, an update factor of 0.2 (20% updates) already causes the total system capacity to be less than a tenth of the nominal capacity.

3.2 A solution

The key to solve this problem is to execute transactions only at their local site, thereby reducing the $w\,(n-1)\,x$ factor in the expressions above. All other sites receive the results of the updates and must only install these updates, which requires significantly less than actually running the transaction. In order to guarantee consistency, the total order established by the to-delivery primitive is used as a guideline to serialize transactions. All sites see the same total order for update transactions. Thus, to guarantee correctness, it suffices for a site to ensure that conflicting transactions are ordered according to the definitive order. Note that transactions can be executed in different orders at different sites if they are not serialized with respect to each other.

When an update transaction T is submitted, it is broadcast to all nodes. This message contains the entire transaction and it is first opt-delivered at all sites (including the local site) which can then proceed to add the corresponding entries in the local queues. Only the local site executes T: whenever T is at the head of any of its queues the corresponding operation is executed on a shadow copy of the data. That way, triggers, consistency constraints and internal read-from dependencies can be observed and aborting the transaction becomes straightforward.

When a transaction is to-delivered at a site, the site checks that the definitive and tentative orders agree. If they agree, the transaction can be committed after its execution has completed. If they do not agree, there are several cases to consider. The first one is when the lack of agreement is with non-conflicting transactions. In that case, the ordering mismatch can be ignored. If the mismatch is with conflicting transactions, there are two possible scenarios. If no local transactions are involved, the transaction can simply be rescheduled in the queues before the transactions that are only opt-delivered but not yet to-delivered. With this to-delivered transactions will then follow the definitive order. If local transactions are involved, the procedure is similar but local transactions must be aborted (because they are executing on the wrong shadow copy) and rescheduled again (by putting them back in the queues in the proper order).

Once a transaction is to-delivered and completely executed the local site broadcasts the commit message containing all updates (also called write set WS). Upon receiving a commit message (which does not need any ordering guarantee), a remote site installs the updates for a certain basic conflict class as soon as the transaction reaches the head of the corresponding queue. When all updates are installed the transaction commits.

3.3 Example

Assume that there are two basic conflict classes C_x, C_y and two sites N and N'. Site N is the master of conflict classes $\{C_x\}$, and $\{C_x, C_y\}$. We denote the conflict class of a transaction T_i by C_{T_i}. Site N' is the master of $\{C_y\}$. Assume there are three transactions, $C_{T_1} = \{C_x, C_y\}$, $C_{T_2} = \{C_y\}$ and $C_{T_3} = \{C_x\}$. That is, T_1 and T_3 are local at N and T_2 is local at N'. The tentative order at N is: T_1, T_2, T_3 and at N' is: T_2, T_3, T_1. The definitive order is: T_1, T_2, T_3. When all the transactions have been opt-delivered, the queues at each site are as follows:

At N:	At N':
$CQ_x = T_1, T_3$	$CQ_x = T_3, T_1$
$CQ_y = T_1, T_2$	$CQ_y = T_2, T_1$

At site N, T_1 can start executing both its operations on C_x and C_y since it is at the head of the corresponding queues. When T_1 is to-delivered the orders are compared. In this case, the definitive order is the same as the tentative order and hence, T_1 can commit. When T_1 has finished its execution, N will send a commit message with all the corresponding updates. N can then commit T_1 and remove it from the queues. The same will be done for T_3 even if, in principle, T_2 goes first in the final total order. However, since these two transactions do not conflict, this mismatch can be ignored. Parallel to this, when N receives the commit message for T_2, the corresponding changes can be installed since T_2 is at the head of the queue CQ_y. Once the changes are installed, T_2 is committed and removed from CQ_y.

At site N', T_2 can start executing since it is local and at the head of its queue. However, when T_1 is to-delivered, N' realizes that it has executed T_2 out of order and will abort T_2, moving it back in the queue. T_1 is moved to the head of both queues. Since T_3 is remote at N', moving T_1 to the head of the queue CQ_x does not require to abort T_3. T_1 is now the first transaction in all the queues, but it is a remote transaction. Therefore, no transaction is executing at N'. When the commit message of T_1 arrives at N', T_1 is executed, committed and removed from both queues. Then, T_2 will start executing again. When T_2 is to-delivered and completely executed, a commit message with its updates will be sent, and T_2 will be removed from CQ_y.

3.4 The NODO Algorithm

The algorithm has been structured according to the different phases in a transaction's execution: a transaction is opt-delivered, to-delivered, completes execution, and commits. As usual, we assume access to the queue is regulated by locks and latches [GR93].

There are also some restrictions on when certain events may happen. For instance, a transaction cannot commit before it has been executed and to-delivered. Each transaction has two state variables to ensure this behavior: The *execution state* of a transaction can be *active* (as soon as it is queued) or *executed* (when its execution has finished). A transaction can only become executed at the site where it is local. The *delivery state* can be *pending* (it has not been to-delivered yet) or *committable* (it has been to-delivered). When a transaction is opt-delivered its state is set to active and pending.

In the following we assume that whenever a transaction is local and the first one in any of its queues, the corresponding operations are submitted for execution.

```
Upon **Opt-delivery** of $T_i$:
  Mark $T_i$ as active and pending
  **For** each conflict class $C_x \in C_{T_i}$
    Append $T_i$ to the queue $CQ_x$
  **EndFor**

Upon **complete execution** of $T_i$:
  **If** $T_i$ is marked as committable **then**
    Broadcast a commit message with $WS_{T_i}$
  **Else**
    Mark $T_i$ as executed
  **EndIf**

Upon **TO-delivery** of $T_i$:
  Mark $T_i$ as committable
  **If** $T_i$ is executed **then**
    Broadcast a commit message with $WS_{T_i}$
  **Else** *($T_i$ has not finished yet or is not local)*
    **For** each $C_x \in C_{T_i}$
      **If** First($CQ_x$) $= T_j \wedge$ Local($T_j$)
          $\wedge$ Pending($T_j$) **then**
        Abort $T_j$
        Mark $T_j$ as active
      **EndIf**
      Schedule $T_i$ before the first transaction
        marked pending in $CQ_x$
    **EndFor**
  **EndIf**

Upon receiving a **commit** message with $WS_{T_i}$:
  **If** not Local ($T_i$) **then**
```

```
    Delay until $T_i$ becomes committable
    **For** each $C_x \in C_{T_i}$
      When $T_i = First(CQ_x)$
        apply the updates of $WS_{T_i}$
          corresponding to $C_x$ to the database
      Remove $T_i$ from $CQ_x$
    **EndFor**
  **Else**
    Remove $T_i$ from all $C_{T_i}$
  **EndIf**
  Commit $T_i$
```

We assume that each of the phases is done in an atomic step. This means, for instance, that adding a transaction to the different queues during opt-delivery or rescheduling transactions during to-delivery is not interleaved with any other action. Note that aborting a transaction simply involves discarding the shadow copy. The transaction is kept in the queues but in different positions.

The commit message is sent once the transaction has been to-delivered and executed at the local site. Nevertheless, the commit message can arrive to other sites before the transaction has been to-delivered at that site. In that case, the definitive order is not yet known, and hence, the transaction cannot commit at that site to prevent conflicting serialization orders. For this reason the processing of the commit message at a remote site is delayed until the corresponding transaction has been to-delivered at that site. Later, when the transaction has been to-delivered and it is at the head of its queues, the updates sent with the commit message are applied to the database and the transaction committed.

4 Reducing Transaction Aborts

4.1 The Problem

In the NODO algorithm, a mismatch between the local optimistic order and the total order involving an executed local transaction results in the local transaction being aborted (if the misordered transactions are conflicting). Note, however, that the abort rate is not necessarily very high since for this to happen, the transactions must conflict, appear in the system at about the same time, and the site where the mismatch occurs must be the local site where the aborted transaction was executing. In all other cases there are no transaction aborts, only reschedulings. Nevertheless, network congestion and high loads can quickly lead to messages not being spontaneously ordered and, thus, to increasing abort rates. To avoid this problem, the NODO algorithm can be optimized by reducing the number of aborted transactions even further.

4.2 A solution

The way to avoid aborting local transactions is to take advantage of the fact that NODO is, to certain extent, a master copy algorithm (remote sites only install updates in the proper order). With this, a local site can unilaterally decide to change the serialization order of two local transactions (i.e., not following the definitive order) and follow the tentative order. This reduces the abort rate, and thus increases throughput and decreases transaction latency. To guarantee correctness, the local site must inform the rest of the sites about the new execution order. No extra messages are needed since this information can be sent in the commit message.

Special care must be taken with transactions that belong to a non-unitary conflict class (e.g., $C_{T_i} = \{C_x, C_y\}$). We will see that a site can only follow the tentative order $T_1 \rightarrow_{OPT} T_2$ instead of the definitive order $T_2 \rightarrow_{TO} T_1$ if T_1's conflict class C_{T_1} is a subset of T_2's conflict class C_{T_2} Otherwise, inconsistencies could occur. We call this new algorithm REORDERING as the serialization order imposed by the definitive order might be changed for the tentative one.

4.3 Example

Assume a database with two basic conflict classes C_x and C_y. Site N is the master of the conflict classes $\{C_x\}$ and $\{C_x, C_y\}$. N' is the master of conflict class $\{C_y\}$. To show how reordering takes place, assume there are three transactions $C_{T_1} = C_{T_3} = \{C_x, C_y\}$, and $C_{T_2} = \{C_x\}$. All three transactions are local to N. The tentative order at both sites is T_2, T_3, T_1. The definitive order is T_1, T_2, T_3. After opt-delivering all transactions they are ordered as follows at both sites:
$QC_x : \ T_2, T_3, T_1$
$QC_y : \ T_3, T_1$

At site N, T_2 and T_3 can start execution (they are local and are at the head of one of their queues). Assume that T_1 is to-delivered at this stage. In the NODO algorithm, T_1 would be put at the head of both queues which can only be done by aborting T_2 and T_3. This abort is, however, unnecessary since N controls the execution of these transactions and the other sites are simply waiting to be told what to do. Thus, N can simply decide not to follow the total order but the tentative order. When such a reordering occurs, T_1 becomes the *serializer transaction* of T_2 and T_3. Note that this can only be done because the transactions are local at N and the conflict classes of T_2 and T_3 are a subset of T_1's conflict class.

Site N' has no information about the reordering. Thus, not knowing better, when T_1 is to-delivered at N', N' will reschedule T_1 before T_2 and T_3 as described in the NODO algorithm. With this, the queues at both sites look at follows:

Queues at site N:	Queues at site N':
$QC_x : \ T_2, T_3, T_1$	$QC_x : \ T_1, T_2, T_3$
$QC_y : \ T_3, T_1$	$QC_y : \ T_1, T_3$

In the meanwhile, at N, T_2 does not need to wait to be to-delivered. Being at the head of the queue and with its serializer transaction to-delivered, the commit message for T_2 can be sent once T_2 is completely executed (thereby reducing the latency for T_2). The commit message of T_2 also contains the identifier of the serializer transaction T_1. With this, when N' receives the commit message, it realizes that a reordering took place. N' will then reorder T_2 ahead of T_1 and mark it committable. N', however, only reschedules T_2 when T_1 has been to-delivered in order to ensure one-copy serializability. The rescheduling of T_3 will take place when the commit message for T_3 arrives, which will also contain T_1 as the serializer transaction. In order to prevent that T_2 and T_3 are executed in the wrong order at N', commit messages are sent in FIFO order (note, that FIFO is not needed in the NODO algorithm).

As this example suggests, there are restrictions to when reordering can take place. To see this, consider three transactions $C_{T_1} = \{C_x\}$, $C_{T_2} = \{C_y\}$ and $C_{T_3} = \{C_x, C_y\}$. T_1 and T_3 are local to N, T_2 is local to N'. Now assume that the tentative order at N is T_3, T_1, T_2 and at N' it is T_1, T_2, T_3. The definitive total order is T_1, T_2, T_3. After all three transactions have been opt-delivered the queues at both sites look as follows:

Queues at site N:	Queues at site N':
$QC_x : T_3, T_1$	$QC_x : T_1, T_3$
$QC_y : T_3, T_2$	$QC_y : T_2, T_3$

Since T_3 is local and it is at the head of its queues, N starts executing T_3. For the same reasons, N' starts executing T_2. When T_1 is to-delivered at N, T_3 cannot be reordered before T_1. Assume this would be done. T_3 would commit and the commit message would be sent to N'. Now assume the following scenario at N'. Before N' receives the commit message for T_3 both T_1 and T_2 are to-delivered. Since T_2 is local, it can commit when it is executed (and the commit is sent to N). Hence, by the time the commit message for T_3 arrives, N' will produce the serialization order $T_2 \rightarrow T_3$. At N, however, when it receives T_2's commit, it has already committed T_3. Thus, N has the serialization order $T_3 \rightarrow T_2$, which contradicts the serialization order at N'.

This situation arises because $C_{T_1} = \{C_x, C_y\}$ is not a subset of $C_{T_3} = \{C_x\}$ and, therefore, T_1 is not a serializer transaction for T_3. In order to clarify why subclasses (i.e., the reordered transaction conflict class is a subset, or subclass, of the one of the serializer transaction) are needed for reordering, assume that T_1 also accesses C_y (with this, $C_{T_3} \subseteq C_{T_1}$). In this case, the queues are:

Queues at site N:	Queues at site N':
$QC_x : T_3, T_1$	$QC_x : T_1, T_3$
$QC_y : T_3, T_1, T_2$	$QC_y : T_1, T_2, T_3$

The subclass property guarantees that T_1 conflicts with any transaction with which T_3 conflicts. Hence, T_1 and T_2 conflict and N' will delay the execution and commitment of T_2 until the commit message of T_1 is delivered. As the commit message of the reordered transaction T_3 will arrive before the one of T_1, T_3 will be committed before T_1 and thus before T_2 solving the previous problem. This means, that both N and N' will produce the same serialization order $T_3 \rightarrow T_1 \rightarrow T_2$.

4.4 REORDERING Algorithm

In general, the REORDERING algorithm is similar to NODO except in a few points (In the following we omit the actions upon opt-delivery since they are they same as in the NODO algorithm).

```
Upon **complete execution** of $T_i$:
  **If** $T_i$ is marked as committable **then**
    Broadcast a commit message $(WS_{T_i,Serializer(T_i)})$
  **Else**
    Mark $T_i$ as executed
  **EndIf**

Upon **to-delivery** of transaction $T_i$:
  **If** $\neg$ Committed($T_i$) $\wedge$ $\neg$ Committable($T_i$) **then**
    *($T_i$ has not been reordered)*
    **If** Local($T_i$) **then**
      **If** $T_i$ is marked executed **then**
        Broadcast a commit message $(WS_{T_i,T_i})$
      **Else** *($T_i$ has not finished yet)*
        Let $AS = \{T_j | C_{T_j} \cap C_{T_i} \neq \emptyset \wedge C_{T_j} \nsubseteq C_{T_i}$
          $\wedge\ \exists C_x \in C_{T_j} \cap C_{T_i}.\ T_j = \text{First}(CQ_x)$
          $\wedge$ Pending$(T_j)$ $\wedge$ Local$(T_j)\}$
        **For** each $T_j \in AS$
          *(abort conflicting transactions that cannot*
          *be reordered)*
          Abort $T_j$ and mark it as active
        **EndFor**
        *(try to reorder transactions)*
        Let $RS = \{T_j | C_{T_j} \subseteq C_{T_i} \wedge T_j \rightarrow_{opt} T_i$
          $\wedge$ Pending$(T_j)$ $\wedge$ Local$(T_j)\}$
        **For** each $T_j \in RS \cup \{T_i\}$ in opt-delivery order
          Mark $T_j$ as committable
          Associate $T_i$ to $T_j$ as serializer transaction
          Schedule $T_j$ before the first transaction
            pending in all $CQ_x | T_j \in C_x$
        **EndFor**
      **EndIf**
    **Else** *(It is a remote transaction)*
      Mark $T_i$ committable
      **For** each conflict class $C_x \in C_{T_i}$
        **If** $T_j = \text{First}(CQ_x) \wedge$ Pending$(T_j)$
            $\wedge$ Local$(T_j)$ **then**
          Abort $T_j$ and mark it as active
        **EndIf**
        Schedule $T_i$ before the first transaction
          marked as pending in queue $CQ_x$
      **EndFor**
    **EndIf**
  **Else** *(the transaction has been reordered)*
```

```
    Ignore the message
  **EndIf**

Upon receiving a **commit** message with $WS_{T_i,T_j}$:
  **If** not Local($T_i$) **then**
    Delay until $T_j$ is committable
    **If** $T_i \neq T_j$ **then**
      Mark $T_i$ as committable
    **EndIf**
  **EndIf**
  **For** each $C_x \in C_{T_i}$
    **If** not Local($T_i$) **then**
      **If** $T_i \neq T_j$ **then**
        Reschedule $T_i$ just before $T_j$ in $CQ_x$
      **EndIf**
      When $T_i$ becomes the first in $CQ_x$
        apply the updates of $WS_{T_i,T_j}$
    **EndIf**
    Remove $T_i$ from $CQ_x$
  **EndFor**
  Commit $T_i$
```

The commit message must now contain the identifier of the *serializer transaction* and follow a FIFO order.

As in NODO, when a transaction T_i is to-delivered, the transaction is marked as committable. At T_i's local site, any non to-delivered local transaction T_j whose conflict class C_{T_j} is a subset of C_{T_i} and that precedes T_i in the queues (reorder set RS) is marked as committable (since now the commit order is no longer the definitive but the tentative order). Thus, it is possible that when a reordered transaction is to-delivered the transaction is already marked as committable or even has been committed. In this case the to-delivery message is ignored. Local non to-delivered conflicting transactions that cannot be reordered and have started execution are aborted (abort set, AS). When the to-delivered transaction is remote, the algorithm behaves as the NODO algorithm.

A remote reordered transaction T_i cannot commit at a site until its serializer transaction is to-delivered at that site. When this happens, T_i is rescheduled before its serializer transaction. The rescheduling together with the FIFO ordering ensure that remote transactions will commit at all sites in the same order in which they did at the local site.

5 Dealing with Failures

5.1 View Changes

In our system, each site has a copy of all data. Thus, each site acts as a primary for the conflict classes it owns and as a backup for all other conflict classes. In the event of site failures, it is just a matter of selecting the new master site for the conflict classes residing in the failed node. The same mechanism could be used for load balancing across nodes by dynamically reassigning the master node of hot-spot conflict classes.

A simple policy is to assign the conflict classes of the failed node to the first site in the new view. That way all nodes have an easy way to know who is the new master for those conflict classes. The new master node, since it also has the transactions the failed node received, will execute and commit those transactions that are still uncommitted after the view change, including those that were already queued when the view change message was delivered. As the new master, it becomes responsible for the execution and sending the commit message with the corresponding updates.

This master replacement algorithm guarantees the availability of transactions in the presence of failures. That is, a transaction will commit as far as there is at least one available site.

5.2 Consistency

The degree of consistency across sites in a failure case depends on the properties of the multicast messages used to send transactions to all sites.

For both algorithms, transaction messages must be uniformly multicast. If that is not the case, i.e., transaction messages are just reliably multicast, inconsistencies may arise. With reliable multicast it is possible for the master of a conflict class to deliver a transaction to itself, execute it, send the commit message and crash. The sites in the new view could then receive the commit message for a transaction they do not know about and, therefore, cannot process in any way. This is not possible with uniform reliable multicast since the master will only execute the transaction after all available sites have received the transaction and thus, they will be able to take over in the event of the master crash.

In the NODO algorithm, commit messages do not have to be uniform since they are a mere confirmation (conflicting transactions are always committed in the total order). Moreover, in order to reduce transaction latency, local transactions can be committed before multicasting the commit message, instead of committing when the commit message is delivered. The worst that can happen in this scenario is that a master commits a transaction and fails before the commit message reaches the other sites. When a new master takes over, it will execute the transaction again, send a new commit message, and the transaction will commit across the system. As the total order is always followed in the NODO algorithm, inconsistencies cannot arise.

In the REORDERING algorithm, commit messages must be uniform since master sites can reorder transactions. If the commit message is not uniform, a master can reorder a transaction, send the commit message and then crash. If the rest of the replicas do not see the commit message, they will use a different serialization order (as the failed node's optimistic order is unknown to the other sites). Uniform message delivery avoids this because the master will not commit a transaction before the commit message has been delivered at all sites.

Upon recovery, a failed site must synchronize its state with that of the available sites. In practice, this can be accomplished by installing a snapshot of the database as of the time of the view change. From now on, the recovered site will receive all the messages delivered after the view change.

6 Correctness

In this section we prove the safety and liveness properties of the two algorithms. The safety property is based on one-copy serializability [BG83]. The liveness property states that any to-delivered transaction will eventually commit.

6.1 Basics

There are several facts that help to prove the correctness of the algorithms. First, since transactions are enqueued (respectively rescheduled) in all corresponding queues in one atomic step, there is no interleaving between transactions. Thus, all sites produce serializable histories. Moreover, the order in which conflicting transactions commit matches the total order. In addition, the proofs that follow assume histories encompassing several views. When talking about correctness (e.g., conflict equivalence between histories), we will refer to the correctness of available nodes.

6.2 Correctness of NODO

Since each site produces serializable histories, it suffices to show that the histories of all sites are conflict equivalent. This can be done by using the total order as a guideline.

Definition 1 (Direct conflict). *Two transactions T_1 and T_2 are in direct conflict if they are serialized with respect to each other, $T_1 \longrightarrow T_2$, and there are no transactions serialized between them: $\nexists T_3 \mid T_1 \longrightarrow T_3 \longrightarrow T_2$.*

Lemma 1 (Total order and Serializability). *Let H_N be the history produced at site N, let T_1, T_2 be two directly conflicting transactions in H_N. If $T_1 \longrightarrow_{TO} T_2$ then $T_1 \longrightarrow_{H_N} T_2$.*

Proof (lemma 1): Assume the lemma does not hold, i.e., there is a pair of transactions T_1, T_2 such that $T_1 \longrightarrow_{H_N} T_2$ but $T_2 \longrightarrow_{TO} T_1$. The fact that T_2 precedes T_1 in the total order means that T_2 was to-delivered before T_1. Since T_1 and T_2 are in direct conflict, they must have at least one conflict class in common. In other words, there was at least one queue where both transactions had entries. If $T_1 \longrightarrow_{H_N} T_2$, then the entry for T_1 must have been ahead in the queue. If T_1 was the first transaction, the NODO algorithm would have aborted T_1 and rescheduled after T_2. If T_1 was not the first in the queue, the NODO algorithm would have put T_2 ahead of T_1 in the queue. In both cases this would result in $T_2 \longrightarrow_{H_N} T_1$ which contradicts the initial assumption. □

Lemma 2 (Conflict equivalence). *For any two sites, N, N', running the* Nodo *algorithm, H_N is conflict equivalent to $H_{N'}$.*

Proof: (lemma 2) From Lemma 1, all pairs of directly conflicting transactions in both H_N and $H_{N'}$ are ordered according to the total order. Thus, H_N and $H_{N'}$ are conflict equivalent since they are over the same set of transactions and order conflicting transactions in the same way. □

Theorem 1 (1CPSR (NODO)). *The history produced by the* Nodo *algorithm is one copy serializable.*

Proof: (theorem 1) From Lemma 2, the histories of all available sites are conflict equivalent. Moreover, they are all serializable. Thus, the global history is one copy serializable. □

6.3 Liveness of NODO

Theorem 2 (Liveness of NODO). *The* Nodo *algorithm ensures that each to-delivered transaction T_i eventually commits in the absence of catastrophic failures.* □

Proof: (theorem 2) The theorem is proved by induction on the position n of T_i in the total order.

Induction Basis: Let T_i be the first to-delivered transaction. Upon to-delivery, each site would place T_i at the head of all its queues. Thus, T_i's master can execute and commit T_i, and then send the commit message to the remote sites. Remote sites will apply the updates and also commit T_i.

Induction Hypothesis: The theorem holds for the to-delivered transactions with positions $n \leq k$, for some $k \geq 1$, in the definitive total order, i.e., all transactions that have at most $k-1$ preceding transactions will eventually commit.

Induction Step: Assume that transaction T_i is at position $n = k+1$ in the definitive total order when it is to-delivered. Each node places T_i in the corresponding queues after any committable transaction (to-delivered before T_i) and before any pending transaction (not yet to-delivered). All committable transactions that are now ordered before T_i have lower positions in the definitive total order. Hence, they will all commit according to the induction hypothesis and be removed from the queues. With this, T_i will eventually be the first in each of its queues and, from the induction basis, eventually commit.

For the induction basis and the induction step, if the master fails before the other sites have received the commit, a new master will reexecute the transaction and resend the commit message. □

6.4 Consistency of NODO

Failed sites obviously do not receive the same transactions as available sites. Let $\mathcal{T}$ be the subset of transactions to-delivered to a node before it failed.

Theorem 3 (Consistency of failed sites). *All transactions, T_i, $T_i \in \mathcal{T}$, that are committed at a failed node N are committed at all available nodes. Moreover, the committed projection of the history in N is conflict equivalent to the committed projection of the history of any of the available nodes when this history is restricted to the transactions in $\mathcal{T}$.* □

Proof: (theorem 3) We have to show that committed transactions at N are also committed at the available sites: For a transaction to be committed anywhere, it must have been to-delivered. Thus, all transactions in $\mathcal{T}$ have been to-delivered and all available sites know about them. If the transaction was not local at N, then N must have received a commit message. The other available sites have either received this commit message (and therefore also commit the transaction) or will commit the transaction when the new master takes over, executes the transaction again, and send the commit message. If they do not receive this second message, it is because the new master also failed. Then another master will take over and repeat the procedure. Since we are assuming there are some available nodes, eventually one of these nodes will become the master and the transaction will commit. If the transaction was local at N, the same argument applies.

The equivalence of histories follows directly from Lemma 1. □

6.5 Correctness of REORDERING

In the Reordering algorithm it is not possible to use the total order as a guideline since a site might decide to reorder transactions. Nevertheless, each site still produces serializable histories. If we can prove that all these histories are conflict equivalent, then the Reordering algorithm produces one copy serializable histories.

We start by proving that transactions not involved in a reordering can not get in between the serializer and the transaction being reordered. Let T_s be the serializer transaction of the transactions in the set $\mathcal{T}_{T_s}$.

Lemma 3 (Reordered). *A reordered transaction T_i is always serialized before its serializer transaction T_s, that is, if $T_i \in \mathcal{T}_{T_s}$ then $T_i \longrightarrow T_s$.*

Proof (lemma 3):

It follows trivially from the algorithm. □

Lemma 4 (Serializer). *In the* Reordering *algorithm, and for all transactions T_i, $T_i \in \mathcal{T}_{T_s}$ there is no transaction T_j, $T_j \notin \mathcal{T}_{T_s}$, such that $T_i \longrightarrow T_j \longrightarrow T_s$.*

Proof (lemma 4): Assume that N is the master site where the reordering takes place. Since T_s is the serializer of T_i, $T_i \longrightarrow_{OPT} T_s$, and $T_s \longrightarrow_{TO} T_i$. Additionally, from Lemma 3 $T_i \longrightarrow T_s$. There are two cases to consider: (a) $T_j \longrightarrow_{TO} T_s$ and (b) $T_s \longrightarrow_{TO} T_j$.

Case (a): since T_j is to-delivered before T_s, N will reorder the queues so that T_j is before T_i, and T_i is before T_s. With T_j ahead of their queues, T_i and T_s cannot be committed until T_j commits. Thus, T_j cannot be serialized in between T_i and T_s.

Case (b): since T_s is to-delivered before T_j and $T_i \notin \mathcal{T}_{T_s}$, all sites will put T_s ahead of T_j in the queues (T_j cannot have committed because it has not yet been to-delivered), if it was not the case. Since $C_{T_i} \subseteq C_{T_s}$, this effectively prevents transactions from getting in between T_i and T_s. Any transaction T_j trying to do so will conflict with T_s and since T_s has been to-delivered before T_j, and T_j has to wait until T_s commits. By that time, T_i will have committed at its master site and its commit message will have been delivered and processed at all sites before the one of T_s. Therefore, the final serialization order will be $T_i \longrightarrow T_s \longrightarrow T_j$. $\square$

Lemma 5 (Conflict Equivalence). *For any two sites, N, N', running the* REORDERING *algorithm, H_N is conflict equivalent to $H_{N'}$.*

Proof: (lemma 5) For two histories to be equivalent, they must have the same transactions and order conflicting transactions in the same way. Since we assume both N and N' to be available, they both see the same transactions. To see that conflicting operations are ordered in the same way there are four cases to consider. Let T_1 and T_2 be two transactions involved in a direct conflict and let C_{T_1} and C_{T_2} be their conflict classes. We can distinguish several cases:

• $C_{T_1} \subseteq C_{T_2}$ and T_1 and T_2 have the same master N''. Assume first $T_2 \longrightarrow_{TO} T_1$:

(a) If N'' reorders T_1 and T_2 with respect to the total order, then, from Lemma 4, no transaction $T_i \notin \mathcal{T}_{T_2}$ can be serialized in between. The commit for T_1 will be sent before the commit for T_2 and in FIFO order. Hence, all sites will then execute T_1 before T_2.

(b) If N'' follows the total order to commit T_1 and T_2, then other sites cannot change this order. The argument is similar to that in Lemma 1 and revolves about the order in which transactions are committed at all sites.

Assume now $T_1 \longrightarrow_{TO} T_2$:

(c) If $C_{T_1} = C_{T_2}$ then cases (a) and (b) apply exchanging T_1 and T_2.

(d) Otherwise $C_{T_1} \subset C_{T_2}$. In this case, N'' has no choice but to commit T_1 and T_2 in to-delivery order (the rules for reordering do not apply). From here, and using the same type of reasoning as in Lemma 1, it follows that all sites must commit T_1 and T_2 in the same order.

• $C_{T_1} \subseteq C_{T_2}$ and either T_1 and T_2 do not have the same master, or $C_{T_1} \cap C_{T_2} \neq \emptyset$ and neither $C_{T_1} \nsubseteq C_{T_2}$ nor $C_{T_2} \nsubseteq C_{T_1}$.

(e) If T_1 or T_2 are involved in any type of reordering at their nodes, Lemma 4 guarantees that there will be no interleavings between the transactions involved in the reordering and the other transaction. Thus, one transaction will be committed before the other at all sites and, therefore, all sites will produce the same serialization order.

(f) If T_1 and T_2 are not involved in any reordering, then upon to-delivery, both of them will be rescheduled in the same (total) order at all sites and then committed. From here it follows that all sites will produce the same serialization order.

- $C_{T_1} \cap C_{T_2} = \emptyset$.

(g) If there is no serialization order between T_1 and T_2 then they do not need to be considered for equivalence.

(h) If there is a serialization order between T_1 and T_2, it can only be indirect. Assume that in N: $T_1 \ldots \longrightarrow T_i \longrightarrow T_{i+1} \longrightarrow \ldots T_2$. Between each pair of transactions in that sequence, there is a direct conflict. Thus, for each pair, the above cases apply and all sites order the pair in the same way. From here it follows that T_1 and T_2 are also ordered in the same way at all sites. □

Theorem 4 (1CPSR (REORDERING)). *The history produced by the* Reordering *algorithm is one copy serializable.*

Proof: (theorem 4) From Lemma 5, all histories are conflict equivalent. Moreover, they are all serializable. Thus, the global history is one copy serializable.

6.6 Liveness of REORDERING

Theorem 5 (Liveness of REORDERING). *The* Reordering *algorithm ensures that each to-delivered transaction T_i eventually commits in the absence of catastrophic failures.* □

Proof: (theorem 5) The proof is similar to the liveness proof of the Nodo algorithm and is an induction on the position n of T_i in the definitive total order.
Induction Basis: Let T_i be the first to-delivered transaction. Upon to-delivery, each remote site will place T_i at the head of all its queues. At the local node, there might be some reordered transactions ordered before T_i and T_i is their serializer. All these can be executed and committed, so that T_i will eventually be executed and committed. Remote sites will apply the updates of the reordered transactions and T_i in FIFO order and hence, they will also commit T_i.

Induction Hypothesis: The theorem holds for the to-delivered transactions with positions $n \leq k$, for some $k \geq 1$, in the definitive total order, i.e., all transactions that have at most $k - 1$ preceding transactions will eventually commit.

Induction Step: Assume that transaction T_i is at position $n = k + 1$ in the definitive total order when it is to-delivered. There are two cases:

a) T_i *is reordered.* This means there is a serializer transaction T_j with a position $n \leq k$ in the total order and T_i is ordered before T_j. Since T_j, according to the induction hypothesis, commits and T_i is executed and committed before T_j at all sites, the theorem holds.

b) T_i *is not a reordered transaction.* T_i will be rescheduled after any committable transaction and before any pending transaction. There exist two types of committable transactions.

i. *Not reordered transactions:* They have a position $n \leq k$ and will therefore commit and be removed from the queues according to the induction hypothesis.

ii. *Reordered transactions:* Each reordered transaction that is serialized by transaction $T_k \neq T_i$ will commit before T_k and T_k will commit according to the induction hypothesis. All transactions $T_j \in \mathcal{T}_{T_i}$ (i.e., T_i is the serializer) are ordered directly before T_i in the queues (Lemma 3). Let T_k be the first not reordered transaction before this set of reordered transactions. T_k will eventually commit according to the induction hypothesis, and therefore also all transactions in $\mathcal{T}_{T_i}$ and T_i itself.

Failures lead to masters reassignment but do not introduce different cases to the above ones. □

6.7 Consistency of REORDERING

Again, let $\mathcal{T}$ be the subset of transactions to-delivered to a node before it failed.

Theorem 6 (Consistency of failed sites). *All transactions, T_i, $T_i \in \mathcal{T}$, that are committed at a failed node N are committed at all available nodes. Moreover, the committed projection of the history in N, is conflict equivalent to the committed projection of the history of any of the available nodes when this history is restricted to the transactions in $\mathcal{T}$.* □

Proof: (theorem 6) Since both transaction and commit messages are sent with uniform reliable multicast, all transactions and their commit messages in $\mathcal{T}$ have been to-delivered to all available sites and can therefore commit at all sites.
To prove the equivalence of histories, the theorem follows directly from Lemma 4. □

7 Conclusions

In spite of the amount of work invested in developing eager data replication protocols, the vast majority of known protocols have never been used in practice. It has only been recently that viable solutions have started to appear based on a tighter synergy between group communication primitives and transaction management techniques. Unfortunatley, to make this approach entirely feasible, it is crucial to demonstrate that it can be improved and optimized for realistic application environments. In this paper, we have proposed two such replication protocols for cluster based applications. These protocols solve the scalability problem of existing solutions and minimize the number of aborted transactions, thereby greatly improving the overall throughput and response time. We are confident that these protocols will form the basis of future database replication techniques. We are currently implementing and experimentally evaluating the protocols and, as part of future work, we will deploy a web farm with a replicated database built upon these protocols.

References

[AAAS97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting Atomic Broadcast in Replicated Databases. In *Euro-Par Conf.*, Passau, Germany, August 1997.

[BG83] P. A. Bernstein and N. Goodman. The Failure and Recovery Problem for Replicated Databases. In *Proc. of 2nd Symp. on Principles of Distributed Computing*, pages 114–122, 1983.

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

[BR93] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of the SIGMOD*, pages 173–182, Montreal, 1996.

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

[HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison Wesley, Reading, MA, 1993.

[KA] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, to appear.

[KA98] B. Kemme and G. Alonso. A Suite of Database Replication Protocols based on Group Communication Primitives. In *Proc. of 18th Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 156–163. IEEE Computer Society Press, 1998.

[KPAS99a] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431. IEEE Computer Society Press, 1999.

[KPAS99b] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. Technical Report 325, ETH Zürich, Department of Computer Science, 1999.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In D. J. Pritchard and J. Reeve, editors, *Proc. of 4th International Euro-Par Conference*, volume LNCS 1470, pages 513–520. Springer, September 1998.

[PS98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In S. Kutten, editor, *Proc. of 12th Distributed Computing Conference*, volume LNCS 1499, pages 318–332. Springer, September 1998.