# Scalable Replication in Database Clusters

M. Patiño-Martínez, R. Jiménez-Peris

Technical University of Madrid
Facultad de Informática
Boadilla del Monte
Madrid, 28660, Spain
{mpatino,rjimenez}@fi.upm.es

B. Kemme, G. Alonso

Swiss Federal Institute of Technology
Department of Computer Science
ETH Zentrum (ETHZ)
CH-8092, Zürich, Switzerland
{kemme,alonso}@inf.ethz.ch

**Abstract.** In this paper, we explore data replication protocols that provide 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 producing results. To further reduce latency, we use an optimistic multicast technique that overlaps transaction execution with total order message delivery. The protocols we present in the paper provide correct executions while minimizing overhead and providing higher scalability.

## 1 Introduction

Conventional algorithms for database replication emphasize consistency and fault tolerance instead of performance [BHG87]. As a result, database designers ignore these algorithms and use *lazy* replication instead, thereby compromising both fault-tolerance and consistency [GHOS96]. A way out of this dilemma [KA98,KA] is to combine database replication techniques with group communication primitives [HT93]. This approach has produced efficient *eager* replication protocols that guarantee consistency and increase fault tolerance. However, in spite of some suggested optimizations [KPAS99,PS98], this new type of protocols still have 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. Transactions, even those over replicated data, are executed at only one site. The other sites only install the final changes. With this, and unlike in existing replication protocols, the aggregated computing power actually increases as more nodes are added. This is a significant advantage in environments with expensive transaction processing (e.g., dynamic web pages). A negative aspect of this protocol is that it might abort transactions in order to guarantee serializability. To reduce the rate of aborted transactions while still providing consistency, we propose a second protocol based on a transaction reordering technique.

The paper is organized as follows. Section 2 introduces the system model. Sections 3 and 4 describe the algorithms. Section 5 discusses fault tolerance aspects. Section 6 contains the correctness proofs. Section 7 concludes the paper.

## 2 System Model

In a replicated database, a group of nodes $N = \{N_1, N_2, ..., N_n\}$, each containing the entire database, communicate by exchanging messages. Sites only fail by crashing (no byzantine failures) and there is always at least one node available.

### 2.1 Communication Model

The system uses various group communication primitives [HT93]. Regarding message ordering, we use a multicast primitive not providing any order, a primitive providing *FIFO order* (messages of one sender are delivered in FIFO order) and one providing a *total order* (all messages are delivered at all sites in the same order). In regard to fault-tolerance, we use both a *reliable delivery service* (whenever a message is delivered at an available site it will be delivered at all available sites) and a *uniform reliable delivery service* (whenever a message is delivered at any faulty or available site it will be delivered at all available sites). We assume a virtual synchronous system, 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.

We use an aggressive version [KPAS99] 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 [KPAS99]. *To-broadcast*(m) broadcasts 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, i.e., partially ordered sets of read and write operations. Two transactions conflict if they access the same data item and at least one of them is a write. A history $H$ of committed transactions 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 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 1-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* [KPAS99]. 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 [KPAS99] 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 a lock table [GR93].

Each conflict class 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* everywhere else.

# 3 Increasing Scalability

## 3.1 The Problem and a Solution

The scalability of data replication protocols heavily depends on the update ratio. To see why, consider a centralized system 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 ($w\,x$) times the number of nodes. From here, the number of transactions that can be processed at each node is $x = t\,(1 + w\,(n-1))^{-1}$. The total capacity of the system is $n$ times that expression which yields, with $t$ normalized to 1, $n\,(1 + w\,(n-1))^{-1}$. 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, 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, $w = 0.2$ (20% updates) results in a system with a tenth of the nominal capacity.

This limitation can be avoided if transactions execute only at one site (the local site) and the other sites only install the corresponding updates. This requires significantly less than actually running the transactions as has been shown in [KA00]. 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. 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 multicast to all nodes. This message contains the entire transaction and it is first opt-delivered at all sites 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. With this, aborting a transaction simply requires to discard the shadow copies. When the transaction commits the shadow copies become the valid versions of the data.

When a transaction is to-delivered at a site, the site checks whether 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 (that have been executed in the wrong order) must be aborted 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.

### 3.2 Example

Assume there are two basic conflict classes $C_x, C_y$ and two sites $N$ and $N'$. $N$ is the master of conflict classes $\{C_x\}$, and $\{C_x, C_y\}$. $N'$ is the master of $\{C_y\}$. We denote the conflict class of a transaction $T_i$ by $C_{T_i}$. 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 at both sites 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 $T_2$ from $N'$, 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 $N'$, $T_1$'s updates are applied, $T_1$ commits and is 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 $CQ_y$.

### 3.3 The NODO Algorithm

The first algorithm we propose, NODO (NOn-Disjoint conflict classes and Optimistic multicast), follows that in [KPAS99]. The algorithm is described according to the different phases in a transaction's execution: a transaction is opt-delivered, to-delivered, completes execution, and commits. We assume access to the queues is regulated by locks and latches [GR93]. There are some restrictions on when certain events may happen. For instance, a transaction can only commit when it has been executed and to-delivered. Waiting for the to-delivery is necessary to avoid conflicting serialization orders at the different sites. 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 its master site. 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.

We assume that each of the phases is done in an atomic step. For instance, 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 itself is kept in the queues but in different positions.

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 **TO-delivery** of $T_i$:
  Mark $T_i$ as committable
  **If** $T_i$ is executed **then**
    Broadcast commit $(WS_{T_i})$
  **Else** *(still active or 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
      pending transaction in $CQ_x$
    **EndFor**
  **EndIf**

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

Upon receiving **commit**$(WS_{T_i})$
  **If** $\neg$ Local $(T_i)$ **then**
    Delay until $T_i$ becomes committable
    **For** each $C_x \in C_{T_i}$
      When $T_i$ becomes the first in $CQ_x$
        Apply the updates of $WS_{T_i}$
          corresponding to $C_x$
        Remove $T_i$ from $CQ_x$
      **EndFor**
    **Else**
      Remove $T_i$ from all $C_{T_i}$
    **EndIf**
    Commit $T_i$

## 4 Reducing Transaction Aborts

In the NODO algorithm, a mismatch between the local optimistic order and the total order may result in a transaction being aborted. The resulting 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 lead to messages not being spontaneously ordered and, thus, to higher abort rates. The number of aborted transactions can be reduced by taking advantage of the fact that NODO is a form of master copy algorithm (remote sites only install updates in the proper order). Thus, a local site can unilaterally decide to change the serialization order of two local transactions (i.e., follow the tentative order instead of the definitive total order), thereby avoiding the abort. To guarantee correctness, the local site must inform the rest of the sites about the new execution order (by appending this information to the commit message). Special care must be taken with transactions that belong to a non basic conflict class (e.g., $C_{T_i} = \{C_x, C_y\}$). 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 $C_{T_2}$ and botho are local transactions. 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.

### 4.1 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 serialize according to the tentative order. This is possible because all transactions involved are local and the conflict classes of $T_2$ and $T_3$ are a subset of $T_1$'s conflict class. When such a reordering occurs, $T_1$ becomes the *serializer transaction* of $T_2$ and $T_3$. $T_2$ does now not need to wait to be to-delivered to commit. 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 $T_1$. The same applies to $T_3$.

Site $N'$ has at the beginning 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. However, when $N'$ receives the commit message of $T_2$, it realizes that a reordering took place (since the commit message contains the information that $T_2$ has been serialized before $T_1$). $N'$ 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 1-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, $T_1$ cannot be a serializer transaction for $T_3$. In order to clarify why subsets (i.e., the conflict class of the reordered transaction is a subset of the conflict class 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 subset 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 $T_3 \rightarrow T_1 \rightarrow T_2$.

### 4.2 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). 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. Note that 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.

Upon **to-delivery** of transaction $T_i$
**If** $\neg$ Committed($T_i$) $\wedge$ Pending($T_i$) **then**
 *($T_i$ has not been reordered)*
 **If** Local($T_i$) **then**
  **If** $T_i$ is marked executed **then**
    Broadcast commit ($WS_{T_i,T_i}$)
  **Else** *($T_i$ has not finished yet)*
    $AS = \{T_j | C_{T_j} \cap C_{T_i} \neq \emptyset \wedge C_{T_j} \nsubseteq C_{T_i}$
      $\wedge \exists C_x \in C_{T_j} \cap C_{T_i}$: $T_j =$ 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)*
    $RS = \{T_j | C_{T_j} \subseteq C_{T_i} \wedge T_j \rightarrow_{opt} T_i$
      $\wedge$ 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
      Mark $T_i$ as serializer ($Ser$) for $T_j$
      Schedule $T_j$ before the first pending transaction 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** $T_j =$ 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** *(transaction has been reordered)*
 Ignore the message
**EndIf**

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

Upon receiving **commit** $WS_{T_i,T_j}$
**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 $WS_{T_i,T_j}$ corresponding to $C_x$
 **EndIf**
 Remove $T_i$ from $CQ_x$
**EndFor**
Commit $T_i$

## 5 Dealing with Failures

In our system, 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, the available sites simply have to select a new master for the conflict classes of the failed node. The new master will also take over the responsibility for all pending transactions for which the failed node was the owner (i.e., where the commit message has not been received by the available sites). Such a 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.

For both algorithms, transaction messages must be uniformly multicast because only then it is guaranteed that the master will only execute and commit a transaction when all sites will receive it, and thus, be able to take over if the master crashes (reliable multicast does not provide this since the master can commit a transaction which the other sites have not yet received).

In the NODO algorithm, commit messages do not need to be uniform. Local transactions can even be committed before multicasting the commit message. The worst that can happen 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 reexecute the transaction and send a new commit message. As the total order is always followed inconsistencies cannot arise.

In the REORDERING algorithm, commit messages must be uniform and the master may not commit the transaction before the commit message is delivered. If the commit message were not uniform, a master could reorder a transaction, send the commit message and then crash. If the rest of the replicas do not see the commit message, they would use a different serialization order (as the failed node's optimistic order is unknown to the other sites).

## 6 Correctness

In this section we prove the correctness (i.e., 1-copy-serializability), liveness, and consistency of the protocols. The proofs assume histories encompassing several group views. Important for both protocols is the fact that transactions are enqueued (respectively rescheduled) in one atomic step. Hence, there is no interleaving between transactions and all sites produce automatically serializable histories. As a result, in order to prove 1-copy-serializability, it suffices to show that all histories are conflict equivalent. Since conflict equivalence requires histories to have the same set of transactions, we refer in the corresponding proofs only to the available sites.

### 6.1 Correctness of NODO

We will show that all sites order conflicting transactions according to the definitive total order.

**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 in NODO).** *Let $H_N$ be the history produced at site $N$, let $T_1$ and $T_2$ 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, 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. However, upon to-delivery of $T_2$, if $T_1$ was the first transaction, NODO would have aborted $T_1$ and rescheduled it after $T_2$. If $T_1$ was not the first in the queue, NODO 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 in NODO).** *For any two sites $N$ and $N'$, $H_N$ 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 in NODO).** *The* NODO *algorithm produces 1-copy-serializable histories.*

**Proof**: (theorem 1) Since the histories of all available nodes are conflict equivalent (lemma 2) and serializable, the global history is 1-copy-serializable. □

### 6.2 Liveness of NODO

**Theorem 2 (Liveness in NODO).** *Each to-delivered transaction $T_i$ eventually commits in the absence of catastrophic failures.*

**Proof**: (theorem 2) The theorem is proved by induction.
*Induction Basis:* Let $T_i$ be the first to-delivered transaction. Upon to-delivery, each site places $T_i$ at the head of all its queues. Thus, $T_i$'s master can execute and commit $T_i$, and then multicast the commit message. 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, as in the induction basis, eventually commit.

In all cases, if the master fails before the other sites have received the commit, the new master will reexecute $T_i$ and resend the commit message. □

### 6.3 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 with NODO).** *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) A transaction $T_i$ can only commit at $N$ when it is to-delivered. Since we use uniform reliable delivery, $T_i$ will also be to-delivered and known at all available sites. If $T_i$ was not local at $N$, then $N$ must have received a commit message from $T_i$'s master. If this master is available for sufficient time all other available sites will also receive the commit message. If the master fails a new master will take over, execute $T_i$ and resend the commit. This procedure will repeat if the new master also fails before the rest of the system receives the commit message. Since we assume 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 2. $\square$

### 6.4 Correctness of REORDERING

In the Reordering algorithm it is not possible to use the total order as a guideline since nodes can reorder local transactions. Thus, we start by proving that transactions not involved in a reordering cannot 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. $\square$

**Lemma 4 (Serializer in REORDERING).** *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) $T_s \longrightarrow_{TO} T_j$.
(a): since $T_j$ is to-delivered before $T_s$, in the queues $T_j$ 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 $T_s$.
(b): since $T_s$ is to-delivered before $T_j$ and $T_j \notin \mathcal{T}_{T_s}$, 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 $T_j$, $T_j$ 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 $T_i \longrightarrow T_s \longrightarrow T_j$. □

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

**Proof**: (lemma 5) We show that two directly conflicting transactions $T_1$ and $T_2$ with conflict classes $C_{T_1}$ and $C_{T_2}$ are ordered in the same way at $N$ and $N'$. We have to distinguish several cases:

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

(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 $T_2$ 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.

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

(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 $T_1$ before $T_2$.

• either $C_{T_1} \subseteq C_{T_2}$ and $T_1$ 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 similar to Lemma 1, both of them will be scheduled in the same (total) order at all sites and then committed.

• $C_{T_1} \cap C_{T_2} = \emptyset$.

(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 in REORDERING).** *The* REORDERING *algorithm produces 1-copy-serializable histories.*

**Proof**: (theorem 4) ¿From Lemma 5, all histories are conflict equivalent. Moreover, they are all serializable. Thus, the global history is 1-copy-serializable. □

### 6.5 Liveness of REORDERING

**Theorem 5 (Liveness in REORDERING).** *Each to-delivered transaction $T_i$ eventually commits in the absence of catastrophic failures.*

**Proof**: (theorem 5) The proof is by induction.
*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 before $T_i$ hence, $T_i$ will be their serializer. All these transactions 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 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 rescheduled before $T_i$.

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 previous point (i). 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 previous point (i), 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. □

### 6.6 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 with REORDERING).** *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. The equivalence of histories, follows directly from Lemma 5. □

## 7 Conclusions

In this paper, we have proposed two replication protocols for cluster based applications. These protocols solve the scalability problem of existing solutions and minimize the number of aborted transactions. 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. For this purpose we will use TransLib [JPAB00], a group-based TP-monitor.

## References

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

[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.

[JPAB00] R. Jiménez Peris, M. Patiño Martínez, S. Arévalo, and F.J. Ballesteros. TransLib: An Ada 95 Object Oriented Framework for Building Dependable Applications. *Int. Journal of Computer Systems: Science & Engineering*, 15(1):113–125, January 2000.

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

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

[KA00] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431, 1999.

[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.