Consistent Database Replication at the Middleware Level

Marta Patiño-Martínez[1,2], Ricardo Jiménez-Peris[1,2]
Facultad de Informática, Universidad Politécnica de Madrid (UPM), Madrid, Spain
{mpatino, rjimenez}@fi.upm.es
and
Bettina Kemme[2]
School of Computer Science, McGill University, Montreal, Canada
kemme@cs.mcgill.ca
and
Gustavo Alonso[2]
Department of Computer Science, Swiss Federal Institute of Technology (ETHZ)
Zürich, Switzerland
alonso@inf.ethz.ch

The widespread use of clusters and web farms has increased the importance of data replication. In this paper, we show how to implement consistent and scalable data replication at the middleware level. We do this by combining transactional concurrency control with group communication primitives. The paper presents different replication protocols, argues their correctness, describes their implementation as part of a generic middleware tool, and proves their feasibility with an extensive performance evaluation. The solution proposed is well suited for a variety of applications including web farms and distributed object platforms.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Distributed Databases*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance Evaluation*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Fault Tolerance. Performance Attributes. Reliability and availability*; C.2.4 [**Computer Systems Organization**]: Computer Communication Networks—*Distributed Systems. Distributed Databases*

General Terms:
Additional Key Words and Phrases: database replication, eager data replication, scalability, middleware.

[1]This work has been partially funded by the Spanish National Science Foundation (*MCYT*), contract number *TIC2001-1586-C03-02*,*TIC2002-10376-E* and by Microsoft Research under contract number *MS-2003-193*.
[2]This work has been partially funded by the European Commission under contract number ADAPT IST-2001-37126.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2004 ACM 0000-0000/2004/0000-0001 $5.00

1. INTRODUCTION

Data replication is a key component of modern data management strategies in computer clusters and web farms. Replication in these systems is used to spread the load across several servers, mask failures of individual servers, and/or increase the processing capacity of the system. In practice, however, replica control (i.e., keeping copies consistent despite updates) is very complex. For instance, most text book replication protocols (e.g., [Bernstein et al. 1987; Weikum and Vossen 2001]) exhibit poor performance and are, therefore, hardly ever used in practice [Gray et al. 1996]. Typical problems in these protocols are lack of scalability, unacceptable response times, high deadlock probability, and very high network traffic [Gray et al. 1996].

A way to avoid these limitations is to use a combination of database techniques and group communication primitives [Pedone et al. 1998; Pedone and Frolund 2000; Kemme and Alonso 2000a; 2000b; Patiño-Martínez et al. 2000; Jiménez-Peris et al. 2002; Stanoi et al. 1998; Holliday et al. 1999; Amir and Tutu 2002; Amir et al. 2002; Rodrigues et al. 2002]. This basic idea can be implemented in different ways. At the database level, and as demonstrated in Postgres-R [Kemme and Alonso 2000a], it is possible to exploit several optimizations to produce a transparent and highly scalable solution that introduces very little overhead. We refer to this as a white box approach because it takes advantage of the database internals. Outside the database and treating the database as a black box, it is possible to provide a solution that is completely database independent [Amir and Tutu 2002; Amir et al. 2002] at the cost of additional overhead when processing transactions (e.g., serial execution and redundant work at all sites).

In this paper, we show how to combine the advantages of both approaches by using a grey box approach. The goal is to achieve the generality of a replication engine external to the database while still being able to exploit certain database specific optimizations. That way, we combine the best of both approaches, generality and a wider flexibility in terms of performance and applications.

For clarity and to illustrate all the design problems involved, the paper proceeds in a stepwise manner. The system model is introduced in Section 2. The first protocol presented (Section 3) aims at minimizing the amount of redundant work in the system. Transactions, even those performing updates on replicated data, are executed at only one site. The other sites in the system only receive and install the new values of changed data items. With this, and unlike in many other data replication protocols, even in update intensive environments the aggregated computing power of the system actually increases as more sites are added. Such an approach has significant practical advantages. 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, executing the transaction, generating the web pages and delivering them to the client. Obviously, if done at all sites, the amount of redundant work can be very high. With this first protocol we show how to avoid redundant work by using a primary copy approach while still maintaining consistency at all times.

A second protocol (Section 4) enhances the first one by minimizing the cost of using group communication. This is done by exploiting an extreme form of optimistic multicast [Kemme et al. 2003] that hides most of the communication overhead behind the transaction execution. The only negative aspect of this protocol is that, like all optimistic protocols, it may abort transactions under heavy load.

The final and main protocol (Section 5) resolves the problem of aborted transactions by

using a reordering technique that reduces to a minimum the probability of having to abort a transaction even when the system is under heavy load. For all the protocols we argue their correctness (Appendix A) and what happens when failures occur (Section 6).

For the second and third protocols, we discuss their implementation as part of a replication engine at the middleware level (Section 7). The approach taken is a grey box approach since it requires from the database system an API which allows to retrieve the updates of a given transaction in form of a write set, and to apply such write set at a remote site. Otherwise, the database system only needs to provide the standard SQL interface. The middleware performs its own concurrency control mechanism to provide 1-copy-serializability (all physical copies appear as one logical copy and although transactions are executed concurrently it appears as if they were executed serially). Application programs containing the SQL statements can be written as usual. They are embedded into the middleware as it is typical for application server environments. The application programmer does not need to be aware of replication. The only requirement is that it must be known in advance which data partitions a particular application program accesses. This can be automatically extracted (using database tables as partitions), and can even be partly determined dynamically when the application is invoked with specific input parameters.

We also provide an extensive performance evaluation that demonstrates the feasibility of the approach (Section 8). It shows that the approach works better than standard distributed locking solutions. Compared to a black box approach, we are able to execute update transactions at one site and only apply the changes at other sites. Our results show that with this, scalability can be achieved even in update intensive environments. We further show that using group communication primitives does not lead to a communication bottleneck, and the implemented concurrency control component works well for a wide range of conflict situations. As such, it does not seem to have a significant disadvantage over a white box approach. A thorough abort analysis compares the performance of two of the algorithms.

Finally, related work is discussed in Section 9, and Section 10 concludes the paper.

2. SYSTEM MODEL AND PROBLEM STATEMENT

We assume an asynchronous system extended with (possible unreliable) failure detectors in which reliable multicast with strong virtual synchrony can be implemented [Friedman and van Renesse 1995]. The system consists of a group of sites $\mathcal{N} = \{N_1, N_2, ..., N_n\}$, also called nodes, which communicate by exchanging messages. Sites only fail by crashing (Byzantine failures are excluded) and do not recover from crash (recovery is studied in [Jiménez-Peris et al. 2002]). We assume there is at least one site that never crashes. Each such site is denoted as an available site. Each site contains a middleware layer and a database layer. The client submits its requests to the middleware layer which performs according operations on the database. The middleware layer instances on the different sites communicate with each other for replica control purposes. The database systems do not perform any communication.

2.1 Communication Model

Sites are provided with a group communication system supporting strong virtual synchrony [Friedman and van Renesse 1995]. Group communication systems provide reliable multicast and group membership services [Birman 1996]. Group membership services provide the notion of view (current connected and active sites). Changes in the composition of a view are eventually delivered to the application. View composition might change due to

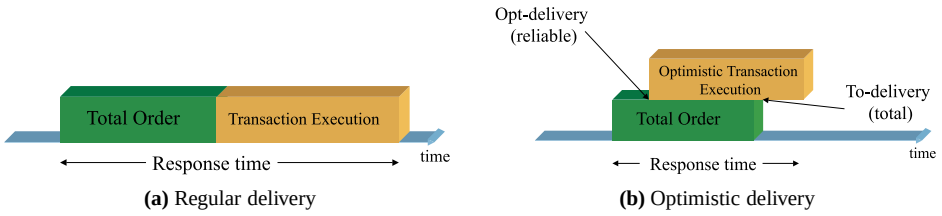
site crashes. Every crashed site is eventually removed from the view. Sites might be falsely suspected to have crashed and are removed from the view. These sites are then forced to commit suicide. We assume a primary component membership [Chockler et al. 2001]. In a primary component membership, views installed by all sites are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one process that survives from the one view to the next one. We say a site is available in a given view V if it delivers V, and also the view following V (or no further view change occurs). Otherwise, the site crashes during V. Strong virtual synchrony ensures that messages are delivered in the same view they were sent (sending view delivery [Chockler et al. 2001]) and that two sites transiting to a new view have delivered the same set of messages in the previous view (virtual synchrony [Chockler et al. 2001]).

Group communication primitives can be classified according to the ordering and reliability guarantees [Hadzilacos and Toueg 1993]. In regard to ordering, the typical primitives are a *simple* multicast (does not provide any ordering guarantees), *FIFO ordering* (all messages sent by a single site are delivered in the order they were sent), and *Total order* (ensures that messages are delivered in the same order at all the sites). In regard to reliability guarantees, *reliable multicast* ensures that all available sites in view V deliver the same messages in V. *Uniform reliable multicast* ensures that a message that is delivered at any site in V (even if it crashes immediately after delivery) will be delivered at all available sites in V. We assume that a multicast message is also delivered to the sender (called self delivery in [Chockler et al. 2001]). Usually, a group communication system also provides support for point-to-point messages.

In the rest of the paper, we use the following primitives. *Multicast(m)/Deliver(m)* denotes the send/delivery of a simple, reliable multicast, *FIFO-Multicast(m)/FIFO-Deliver(m)* the send/delivery of a FIFO, uniform reliable multicast, and *UNI-Send(m)/UNI-Deliver(m)* the send/delivery of a reliable point-to-point message.

Additionally, we assume the existence of an optimistic total order multicast protocol specially tailored for transaction processing [Kemme et al. 2003]. Messages are optimistically delivered as soon as they are received and before the definitive ordering is established. In the traditional approaches (see Fig. 1 (a)), first the total order of a transaction is determined, then the transaction is executed. In our approach, a message is OPT-delivered as soon as it is received from the network and before the definitive ordering is established. Transaction processing can start directly after the OPT-delivery. With this, the execution of a transaction overlaps with the calculation of the total order (Fig. 1 (b)), and response times are only then affected by the delay of the total order multicast if establishing the total order takes longer than executing the transaction (what is usually never the case). If the initial order is the same as the definitive order, the transactions can simply be committed. If the definitive order is different, additional actions have to be taken to guarantee consistency.

This optimistic multicast is defined by three primitives [Kemme et al. 2003]. *TO-Multicast*(m) multicasts the message m to all the sites in the system. *OPT-deliver*(m) delivers message m optimistically to the application, with the same semantics as a simple, reliable multicast (no ordering guarantees). *TO-deliver(m)* delivers m definitively to the application with the same semantics as a total order, uniform reliable multicast. That is, messages can be OPT-delivered in a different order at each available site, but are TO-delivered in the same total order at all available sites. Furthermore, this optimistic multicast primitive ensures that no site TO-delivers a message before OPT-delivering it. A sequence



(a) Regular delivery **(b)** Optimistic delivery

Fig. 1. Reduced response time through optimistic message delivery

of OPT-delivered messages is a *tentative order*. A sequence of TO-delivered messages is the *definitive order* or total order.

2.2 Transaction Model, Correctness Criteria, and Transaction Execution

Clients interact with the system middleware by issuing application-oriented functions (e.g., bookFlight, purchaseBook, etc.). Each of these functions is a pre-implemented application program consisting of several database operations. At the time the request is submitted with a given set of parameters, both the set of operations to be executed, and the data items to be accessed within these operations is known. Such execution model reflects quite well the current use of application servers. Each application program is executed within the context of a transaction. From now on, we only use the term transaction instead of application program. We define a transaction as a partially ordered set of read (r) and write (w) operations on physical data item copies. A transaction can either be read-only (only read operations), or an update transaction (at least one write operation).

For replicated databases, the correctness criterion is one-copy-serializability [Bernstein et al. 1987]. Despite the existence of multiple copies, each data item must appear as one logical-copy (*1-copy-equivalence*), and the execution of concurrent transactions is coordinated so that it is equivalent to a serial execution over the logical copy (*serializability*). In order to achieve 1-copy-serializability we have to guarantee that the global history describing the execution of transactions is equivalent to a history produced by a serial execution of the same transactions. A history H of committed transactions indicates the order in which operations on data copies are executed. It includes the partial execution order within transactions. Furthermore, any two conflicting operations must be ordered. Two operations conflict, if they are from different transactions, access the same data item, and at least one is a write operation. Non-related operations can be executed in parallel, hence, a history is a partial order. Each site has a local history indicating only operations on the local copies. The global history of the system is the union of local histories. A history H of committed transactions is serial if for any two transactions $T1$ and $T2$, either all operations of $T1$ are ordered before all operations of $T2$ or vice versa. Two histories H_1 and H_2 are conflict equivalent if they have the same set of operations and order conflicting operations in the same way. A history H is serializable if it is conflict equivalent to some serial history [Bernstein et al. 1987].

In order to achieve a global, serializable history, the system applies concurrency control. In this paper, concurrency control for update transactions is based on conflict classes [Bernstein et al. 1980; Skeen and Wright 1984]. A *basic conflict class* represents a partition of the data. How to partition the data is application dependent. In a simple case, there could be a class per table. If the application is well structured, other granularities are

possible. For instance, by submitting a purchase order the system exactly knows the items to be bought, and the client who performs the update. Hence, record-level conflict classes can be implemented. If each application program accesses different parts of the database (e.g. users that do not share data), then one can use one conflict class per application program. That is, conflict classes have the same or a coarser granularity than the concurrency component of the database system (which usually works on record-level). In summary, 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 the first algorithm, we propose each transaction must access a single basic conflict class (e.g., C_x). For the rest of the algorithms we generalize this model and allow a transaction to access a *compound conflict class*. A compound conflict class is a non-empty set of basic conflict classes (e.g., $\{C_x, C_y\}$). We denote as C_T T's conflict class (either a basic or compound), and assume that C_T is known in advance (as mentioned above).

The middleware layer implements the above concurrency control. At each site there is 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 queue(s) of the basic conflict class(es) it accesses. This concurrency control mechanism is a simplified version of the lock table used in databases [Gray and Reuter 1993]. In a lock table there is a queue for each data item and locks can be shared or exclusive, whilst in our approach each queue corresponds to an arbitrary set of data items (i.e., a basic conflict class), and access to a particular basic conflict class is always exclusive. We assume the existence of a latch that allows a transaction to access several basic conflict classes in an atomic step if necessary.

Our algorithms assume that each conflict class (basic or compound) has a *master* site. Conflict classes are statically assigned to sites, but in case of failures, they are reassigned to different sites. There are several mechanisms to assign conflict classes in an efficient way. In Section 8.1, we will present an assignment based on hashing. We say a transaction T is *local* to the master site of its conflict class C_T, and is *remote* to the rest of the sites[3].

Transaction execution for all algorithms is roughly as follows. The client submits a transaction to any site using UNI-send. This site immediately forwards the message to all sites. All sites append T to the queues of the basic conflict classes T accesses. Only the master executes the transaction whenever it is the first in all queues. It executes both read and write operations on the local database. Then, it multicasts the updates performed by T to the remote sites. Remote sites only apply these updates instead of reexecuting the entire transaction. In order for the middleware to send and apply writesets, we assume that the underlying database provides two services. One to obtain the write set of a transaction and another one to apply the write set. Executing local transactions and applying the updates of remote transactions must be controlled such as to guarantee 1-copy-serializability. In particular, transactions can execute concurrently if they do not have any common basic conflict class, however, as soon as they share one basic conflict class the execution of the

[3]We would like to note that the algorithms do also work correctly if conflict classes do not have masters. The only requirement for correctness is that each transaction is local at one site and remote at the other sites, and that each site can decide independently whether an incoming transaction is local or remote. Having each conflict class a master and letting a transaction T be local at the master of C_T is just one option to decide who is going to execute the transaction; this option is advantageous because by executing all transactions on a data partition on one site we increase the probability that data to be accessed already resides in main memory, and hence, speed up transaction processing.

Queue Management	
CQ_x	FIFO queue of transactions that want to access conflict class C_x
First(CQ_x)	returns first transaction of queue CQ_x of conflict class C_x (without removing the transaction)
Append(T, CQ_x)	appends T to queue CQ_x of conflict class C_x
Remove(T, CQ_x)	removes T from queue CQ_x of conflict class C_x
Transaction information (T)	
exState	describes the execution state of the transaction; states differ for the different algorithms
delState	describes which messages have been delivered; states differ for the different algorithms
C_T	for the DISCOR algorithm: the basic conflict class T accesses; for NODO and REORDERING: the set of basic conflict classes T accesses
WS_T	write set of transaction T
Local(T)	returns TRUE on the site that is master of C_T, returns FALSE otherwise

Table I. Data structures and general methods

two transactions will be serial according to their order in the corresponding queue. Note that deadlocks cannot occur since a transaction is appended to all basic conflict classes it accesses at transaction start.

Read-only transactions are only executed at the site they are submitted to. We assume that they read data from a snapshot, and hence, do not need to be isolated in regard to update transactions. If the underlying database system does not support such semantics, our model and algorithms are quite simple to extend to integrate local, serializable execution of read-only transactions.

In the following, we present three algorithms (DISCOR, NODO, and REORDERING), that provide 1-copy serializability using the transaction model and execution pattern as described above. Basic data structures and methods used by all of the algorithms are depicted in Table I. We assume that once a site receives the first message regarding a transaction T, the information regarding T is stored in a hash table. Once all messages regarding these transactions have been delivered and T is completely terminated, T is removed from the hash table. The maintenance of this hash table is not further described in the algorithms of the next sections.

3. INCREASING SCALABILITY

3.1 DISCOR Protocol

In the DISCOR (DISjoint COnflict classes and Reliable multicast) algorithm each transaction T belongs to a single basic conflict class C_T. In DISCOR, the execution and delivery states of a local transaction are irrelevant. A remote transaction can have the execution state *executable*, and the delivery state *delivered*. Before a state is explicitly assigned, it is considered NIL.

Figure 2 depicts the DISCOR algorithm. It only considers update transactions since we assume read-only transactions to read from a snapshot. The algorithm has been structured according to where execution takes place (master/remote), and according to different events during the processing of a transaction. In this algorithm, we assume that queues are protected objects, and at most one operation can be active on a queue at any given time.

An update transaction T_i can be submitted to any site (Figure 2(a)). This site will sim-

ply multicast the transaction to all sites without any ordering requirement. Execution is different at the master and the remote sites. The master (Figure 2(b)) adds the transaction to the queue of the basic conflict class C_T. Once T is at the head of its queue the corresponding operations are executed. After the execution has completed, the master site multicasts a commit message containing write set WS_T using FIFO ordering, and commits the transaction. At the remote sites (Figure 2(c)), when the transaction is delivered, it is marked delivered and added to the corresponding queue unless the commit message arrived earlier. The execution order is determined by the order in which commit messages are delivered. Since transactions only access one basic conflict class, conflicting transactions have always the same master. Hence, all sites see the same FIFO order for commit messages of conflicting transactions. Thus, to guarantee correctness, it suffices for a site to ensure that conflicting transactions are ordered according to this FIFO order. Hence, upon delivery of the commit message of transaction T, T is marked executable, and the queue of T's conflict class is reordered, such that already executable transactions (commit message delivered) are ordered before T and active transactions (commit message still missing) are ordered after T. When a transaction is at the head of a queue and the commit message has been delivered, the write set can be applied and the transaction can commit[4].

3.2 Example

Assume that there are two basic conflict classes C_x, C_y and two sites N and N'. N is master of conflict class $\{C_x\}$, and N' is master of $\{C_y\}$. Assume there are three update transactions, $C_{T_1} = \{C_x\}$, $C_{T_2} = \{C_y\}$, 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 delivery order at N is: T_1, T_2, T_3 and at N' is: T_3, T_1, T_2. When all the transactions have been delivered, the queues at each site are as shown in Figure 3. In this figure and all successive examples, the head of the queues is on the left.

At site N, T_1 can start executing its operations on C_x since it is local and it is at the head of the corresponding queue. When T_1 has finished its execution, N will multicast a commit message with all the corresponding updates. When this message is delivered at N, T_1 is committed and removed from the queue. The same will be done for T_3. Site N' performs the same steps for T_2.

When N delivers the commit message for T_2, the corresponding changes can be installed. Once this happens, T_2 is committed and removed from CQ_y. When the commit message of T_1 is delivered at N', T_1 is not at the front of the queue. At N' T_3 was delivered before T_1. Since the commit message indicates the serialization order, T_1 is moved to the front of the queue. Then the updates of T_1 are installed, and T_1 is removed from CQ_x.

4. INCREASING FLEXIBILITY

The DISCOR algorithm requires transactions to be confined to a single conflict class. This is a severe restriction since it assumes the replicated data and the workload are perfectly partitionable. In this section we propose a new algorithm, NODO (NOn-Disjoint conflict classes and Optimistic multicast), that extends the DISCOR algorithm by allowing transactions to access compound conflict classes.

[4] An implementation of such algorithm does not necessarily need to use queues. To be more efficient, incoming transactions could be stored in a hash table and only upon delivery of the commit message, transactions are applied in FIFO order. We use the queues to use a similar structure for all the algorithms presented.

Upon **UNI-deliver**(T_i)
Multicast(T_i)

(a) Transaction Submission to any site

Upon **deliver**(T_i)
Append($T_i, CQ_{C_{T_i}}$)

Upon **First**($CQ_{C_{T_i}}$)= T_i
Submit execution of T_i

Upon **complete execution** of T_i
FIFO-Multicast(commit,WS_{T_i})

Upon **FIFO-Deliver**(commit,WS_{T_i})
Submit commit of T_i
Remove($T_i, CQ_{C_{T_i}}$)

(b) Master site

Upon **deliver**(T_i)
T_i.delState:= delivered
If T_i.exState = NIL **then**
Append($T_i, CQ_{C_{T_i}}$)
EndIf

Upon **First**($CQ_{C_{T_i}}$)= T_i
$\wedge$ T_i.exState = **executable**
Apply the updates of WS_{T_i}

Upon **FIFO-Deliver**(commit,WS_{T_i})
T_i.exState:= executable
If T_i.delState = NIL **then**
Append($T_i, CQ_{C_{T_i}}$)
EndIf
Reorder T_i within $CQ_{C_{T_i}}$
after the last executable transaction

Upon **complete application** of WS_{T_i}
Submit commit of T_i
Remove($T_i, CQ_{C_{T_i}}$)

(c) Remote site

Fig. 2. DISCOR

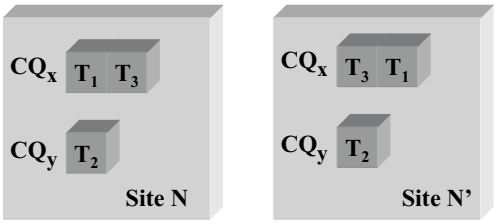


Fig. 3. Example for the DISCOR algorithm

4.1 NODO Protocol

This time, when an update transaction T is submitted, the TO-multicast is used to forward the transaction to all sites. The idea is to start transaction execution upon OPT-delivery, but to use the total order established by the TO-delivery 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. Since the execution order is not important for non-conflicting transactions, they can be executed in different orders (or in parallel) at different sites.

Upon **UNI-deliver**(T_i)
TO-multicast(T_i)

(a) Transaction Submission (to any site)

Upon **OPT-deliver**(T_i)
T_i.delState:= pending
T_i.exState:= executable
For each $C_x \in C_{T_i}$: Append(T_i, CQ_x)

Upon $\forall C_x \in C_{T_i}$, **First**(CQ_x)= T_i
$\wedge$ T_i.exState = **executable**
Submit T_i for execution

Upon **complete execution** of T_i
T_i.exState:= executed
If T_i.delState = committable **then**
Multicast(commit,WS_{T_i})
EndIf

Upon **Deliver**(commit, WS_{T_i})
Submit commit of T_i
For each $C_x \in C_{T_i}$: Remove(T_i, CQ_x)

Upon **TO-deliver**(T_i)
T_i.delState:= committable
If T_i.exState = executed **then**
Multicast(commit,WS_{T_i})
Else *(still executable)*
For each $C_x \in C_{T_i}$
If $\exists T_j$ | (First(CQ_x) = $T_j \wedge$ Local(T_j)
$\wedge$ T_j.delState = pending) **then**
Submit abort of T_j
T_j.exState:= active
EndIf
Reorder T_i within CQ_x
before the first pending transaction
EndFor
EndIf

Upon **complete abort** of T_i
T_i.exState:= executable

(b) Master site

Upon **OPT-deliver**(T_i)
T_i.delState := pending
For each $C_x \in C_{T_i}$: Append(T_i, CQ_x)

Upon T_i.delState = **committable** $\wedge$
T_i.exState = **executable**
$\wedge \forall C_x \in C_{T_i}$ **First**(CQ_x)= T_i
Apply the updates of WS_{T_i}

Upon **complete application** of WS_{T_i}
Submit commit of T_i
For each $C_x \in C_{T_i}$: Remove(T_i, CQ_x)

Upon **TO-deliver**(T_i):
T_i.delState:= committable
For each $C_x \in C_{T_i}$
If $\exists T_j$ | (First(CQ_x) = $T_j \wedge$ Local(T_j)
$\wedge$ T_j.delState = pending) **then**
Submit abort of T_j
T_j.exState:= active
EndIf
Reorder T_i within CQ_x
before the first pending transaction
EndFor

Upon **Deliver**(commit, WS_{T_i})
T_i.exState:= executable

(c) Remote site

Fig. 4. NODO

Figure 4 presents the NODO algorithm. For NODO, the execution state $exState$ of a transaction can be $active$ (cannot yet start execution), $executable$ (may start execution) or $executed$. The delivery state $delState$ can be $pending$ (a transaction is OPT-delivered but not yet TO-delivered), or $committable$ (a transaction is TO-delivered). Within this algorithm we assume that all queue related operations triggered upon a certain event are executed as an atomic step. This can be achieved, e.g., by acquiring a short lived lock (mutex) on the entire queue table. When a transaction T is OPT-delivered at site N, it is added to the queues of all basic conflict classes contained in C_T. Only the master site of C_T executes T: whenever T is at the head of all of its queues the transaction is submitted for execution. When a transaction T is TO-delivered at N, N checks that the definitive and tentative orders agree. If they agree, T 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, T can simply be rescheduled in the queues before the transactions that are only OPT-delivered but not yet TO-delivered (that is the queue is reordered to reflect the total order determined by TO-delivery). If local transactions are involved, the procedure is similar but a local pending transaction T' that might already have started execution (it is the first in its queue), must be aborted. Note that the algorithm does not wait to reschedule until the abort is complete but schedules T immediately before T'. Hence, T might be submitted to the database before T' has completed the abort. However, since the database has its own concurrency control, we have the guarantee that T cannot access any data item T' has written before T' undoes the change. An aborted transaction can only be resubmitted for execution once the abort is complete.

Once a transaction is TO-delivered and completely executed the local site multicasts the commit message including the write set using simple, reliable multicast. Hence, 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 write set is applied to the database and the transaction committed.

4.2 Examples

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 $\{C_x, C_y\}$, and N' is master of $\{C_y\}$. 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. Fig. 5 shows the queues at each site just after all transactions have been OPT-delivered.

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 multicast 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

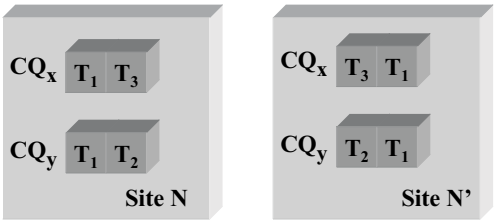


Fig. 5. Example of NODO

these two transactions do not conflict, this mismatch can be ignored. Parallel to this, when N delivers 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 not executed 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 updates are applied, and then $T1$ is 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 multicast, and T_2 will be removed from CQ_y.

5. REDUCING TRANSACTION ABORTS

5.1 REORDERING Protocol

In the NODO algorithm, a mismatch between the local optimistic order and the total order might lead to the abort of a local transaction (if the local transaction already started execution, and the disordered transactions conflict). The way to avoid aborting local transactions is to take advantage of the fact that we follow a master copy approach (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 but 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 access a compound conflict class (e.g., $C_{T_i} = \{C_x, C_y\}$). We will see that a site can only follow the tentative order T_2, T_1 instead of the definitive order T_1, T_2 if T_2's conflict class C_{T_2} is a subset of T_1's conflict class C_{T_1} and both transactions are local. Otherwise, inconsistencies might occur. When reordering can take place (i.e., $C_{T_2} \subseteq C_{T_1}$), T_1 becomes the *serializer transaction* of T_2, and T_2 is a *reordered* transaction. We call this new algorithm REORDERING as the serialization order imposed by the definitive order might be changed for the tentative one.

REORDERING has the same execution and delivery states as NODO. For the master site, the REORDERING algorithm basically does the same as NODO except for the time a transaction is TO-delivered. Figure 6 shows the actions for the master. It only provides the

Upon **TO-deliver**(T_i)
  If T_i.delState = committable **then**
    Ignore message (was reordered)
  Else
    If T_i.exState = executed **then**
      T_i.delState:= committable
      FIFO-Multicast(commit,WS_{T_i}, T_i, {})
    Else *(still executable)*
      Reorder(T_i)
    EndIf
  EndIf

Upon **complete execution** of T_i
  If T_i.delState = committable **then**
    FIFO-Multicast(commit,WS_{T_i}, $Ser(T_i)$, BT_{T_i})
  EndIf
  T_i.exState:= executed

Upon **FIFO-deliver**(commit, WS_{T_i}, $Ser(T_i)$, BT_{T_i})
  Submit commit of T_i
  For each $C_x \in C_{T_i}$: Remove(T_i, CQ_x)

Function Reorder(T_i)
$AS = \{T_j | C_{T_j} \cap C_{T_i} \neq \emptyset \wedge C_{T_j} \not\subseteq C_{T_i}$
  $\wedge \forall C_x \in C_{T_j} : T_j = \text{First}(CQ_x)$
  $\wedge\ T_j$.delState = pending $\wedge$ Local(T_j)$\}$
For each $T_j \in AS$
  Submit abort of T_j
  T_j.exState:= active
EndFor
$RS = \{T_j | C_{T_j} \subseteq C_{T_i} \wedge T_j \rightarrow_{opt} T_i$
  $\wedge\ T_j$.delState = pending $\wedge$ Local(T_j)$\}$
For each $T_j \in RS \cup \{T_i\}$
  in opt-delivery order
    T_j.delState:= committable
    $Ser(T_j) := T_i$ *(T_i is serializer of T_j)*
    For each $C_x \in C_{T_i}$
      Reorder T_i within CQ_x
        before the first pending transaction
    EndFor
    $BT_{T_j} = \{T_k | C_{T_k} \cap C_{T_j} \neq \emptyset \wedge T_k \in RS$
        $\wedge T_k \rightarrow_{opt} T_j\ \}$
    If T_j.exState = executed **then**
      FIFO-Multicast(commit,WS_{T_j}, T_i, BT_{T_j})
    EndIf
EndFor

Fig. 6. REORDERING for a master site

steps that are different from NODO. When a transaction T is TO-delivered we first check whether it is already marked committable or not. In the latter case, T was reordered, and nothing has to be done. Otherwise, if it is completely executed (also meaning it is the first in all queues), we multicast the write set using FIFO order as in DISCOR. If T was not the first in all queues we check for two sets of transactions. AS is the set of transactions that must be aborted, and RS is the set that can be reordered and serialized before T. A transaction in AS is local and pending was OPT-delivered before T, accesses at least one basic conflict class also accessed by T, and at least one conflict class not accessed by T, and has already started execution. A transaction in RS is also local and pending, and was OPT-delivered before T, but in contrast to AS, it accesses a subset of the conflict classes accessed by T. For any two transactions T_1 and T_2 in RS that conflict (their conflict classes intersect), we must make sure that they are executed in the same order at the master and the remote sites. This order is the order they are OPT-delivered at the master (in the algorithm, we denote $T_1 \rightarrow_{OPT} T_2$ if T_1 is OPT-delivered before T_2). Upon complete execution of transaction T, the master multicasts a commit message as in NODO. This time, the commit message also contains the identifier of the *serializer transaction* $Ser(T)$ of T ($Ser(T) = T$, if T was not reordered), and is multicast in FIFO order. Furthermore, it contains the set BT_T including the identifiers of all transactions that are also reordered by $Ser(T)$, and that conflict with T and must be executed before T (because they were OPT-delivered before T). The rest of the algorithm for master is the same as in NODO.

For a remote site, the REORDERING algorithm is depicted in Figure 7. Care has to be taken that conflicting transactions are applied in the same order as at the master. This

Upon **OPT-deliver**(T_i)
  If T_i.delState = NIL **then**
    T_i.delState:= pending
  EndIf
  If T_i.exState = NIL **then**
    For each $C_x \in C_{T_i}$: Append(T_i, CQ_x)
  EndIf

Upon **TO-deliver**(T_i)
  If T_i.delState = committable **then**
    Ignore the message (*was reordered*)
  Else
    T_i.delState:= committable
    For each $C_x \in C_{T_i}$
      If $\exists T_j \mid$ (First(CQ_x) = $T_j \wedge$ Local(T_j)
        $\wedge$ T_j.delState = pending) **then**
          Submit abort of T_j
          T_j.exState := active
      EndIf
      Reorder T_i within CQ_x
        before the first pending transaction
    EndFor
  EndIf

Upon **FIFO-deliver**(commit, WS_{T_i}, $Ser(T_i)$, BT_{T_i})
  T_i.exState:= executable
  If T_i.delState = NIL **then**
    For each $C_x \in C_{T_i}$: Append(T_i, CQ_x)
  EndIf

Only for T_i with $T_i \neq Ser(T_i)$:
Upon T_i.exState = **executable**
  $\wedge$ Ser(T_i).delState = **committable**
  $\wedge$ $\forall T_j \in BT_{T_i}, T_j$.delState = **committable**
    T_i.delState := committable
    For each $C_x \in C_{T_i}$
      Reorder T_i just before $Ser(T_i)$ in CQ_x
    EndFor

Upon T_i.delState = **committable**
  $\wedge$ T_i.exState = **executable**
  $\wedge$ $\forall C_x \in C_{T_i}$: **First**(CQ_x)= T_i
  $\wedge$ $\forall T_j \in BT_{T_i}$: T_j.exState = **executed**
    Apply updates of WS_{T_i}

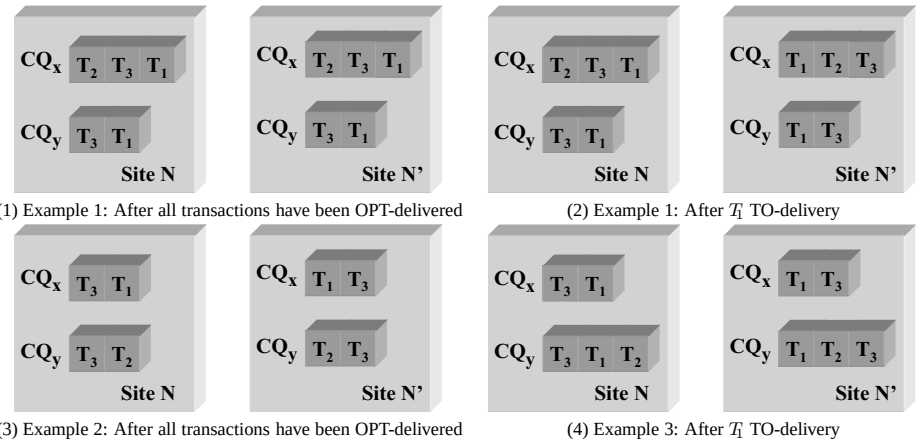
Upon **complete application** of WS_{T_i}
  Submit commit of T_i
  T_i.exState:= executed
  For each $C_x \in C_{T_i}$: Remove(T_i, CQ_x)

Fig. 7. REORDERING for a remote site

requires a bit more careful setting of execution and delivery states. In general, we set a transaction T to be executable when the commit message arrives. We set T committable if it is TO-delivered, or if it was reordered, and the serializer transaction and all transactions that were reordered by the same serializer, and were OPT-delivered at the master before T, have been set committable. The write set of transaction T can be applied when T is executable, committable, at the head of all its queues, and it is guaranteed that all transactions reordered before T have already been reordered at the remote site, and have been committed. Since the execution order is now not necessarily determined by the definitive order, and transactions at remote sites might commit before they are even OPT-delivered (it might be enough that the commit message was delivered and the serializer was TO-delivered), we have to be careful when to exactly set states, and to append the transaction to the queues before applying the write set.

5.2 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 shown in Fig. 8(1).



(1) Example 1: After all transactions have been OPT-delivered (2) Example 1: After T_1 TO-delivery

(3) Example 2: After all transactions have been OPT-delivered (4) Example 3: After T_1 TO-delivery

Fig. 8. Examples of REORDERING

At site N, T_2 can start execution (it is local and at the head of all its 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. 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 does 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 as shown in Fig. 8(2).

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 multicast 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' delivers 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 FIFO-multicast, and $BT_{T_3} = \{T_2\}$ indicating, that T_2 must be executed before T_3.

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 shown in Fig. 8(3).

Since T_3 is local and 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 multicast to N'. Now assume the following scenario at N'. Before N' delivers 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 multicast 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 delivers 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 $C_{T_3} = \{C_x, C_y\}$ is not a subset of $C_{T_1} = \{C_x\}$ 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 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 look like shown in Fig. 8(4).

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 commit 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 $T_3 \rightarrow T_1 \rightarrow T_2$.

6. VIEW CHANGES

So far, failures have not been considered. In our system a site acts as a primary copy for the conflict classes it owns and as a backup for all other conflict classes. In the event of site failures, and the delivery of a view change, it is just a matter of selecting the new master site for the conflict classes residing in the failed site. A simple policy is to assign the conflict classes of the failed site to the first site in the new view. That way all sites have an easy way to know who is the new master for those conflict classes. Other, more sophisticated reassignments are possible.

For each transaction T of the old master for which the new master had delivered the transaction but not the commit message before the view change the new master will execute and commit T. It will also multicast T's commit message. The old master might have delivered a transaction and started its execution but no other site delivered the transaction. However, in this case it is guaranteed that the old master has not committed this transaction. This is due since at least one message is sent using uniform reliable multicast. Hence, providing such simple master takeover in case of view changes, all available sites commit the same transactions and failed sites commit a prefix of these transactions. As such, we avoid having a 2-phase commit protocol (2PC) at the end of a transaction as traditional solutions do. With this we do not only avoid the logging and communication overhead of 2PC but also avoid the requirement that a site can only commit a transaction once it is executed at all sites. In our system, it is enough that a site knows the other sites have delivered the messages and hence, will eventually commit the transaction (unless they fail). Please, refer to the proof section for details.

7. IMPLEMENTATION

In this section we describe the architecture of the middleware replication system. Both the NODO and the REORDERING protocols are implemented. The middleware has been implemented using the C programming language. Multithreading and multiprocessing are heavily used along the different components of the system to provide an optimal degree of concurrency. Communication among the threads of the same process takes place through shared memory and semaphores, whilst communication between different processes is based on Unix sockets.

7.1 Architecture

At each site there is an instance of the middleware and an instance of the database. The middleware is located between the clients submitting transactions and the database (Fig. 9). Clients submit requests to execute predefined application programs (the requests can contain parameters). An application program can have several SQL statements to be executed within the database. The middleware takes care of executing each application program within the context of a transaction. There are three main components in the middleware, the queue manager, the database interceptor, and the communication manager.

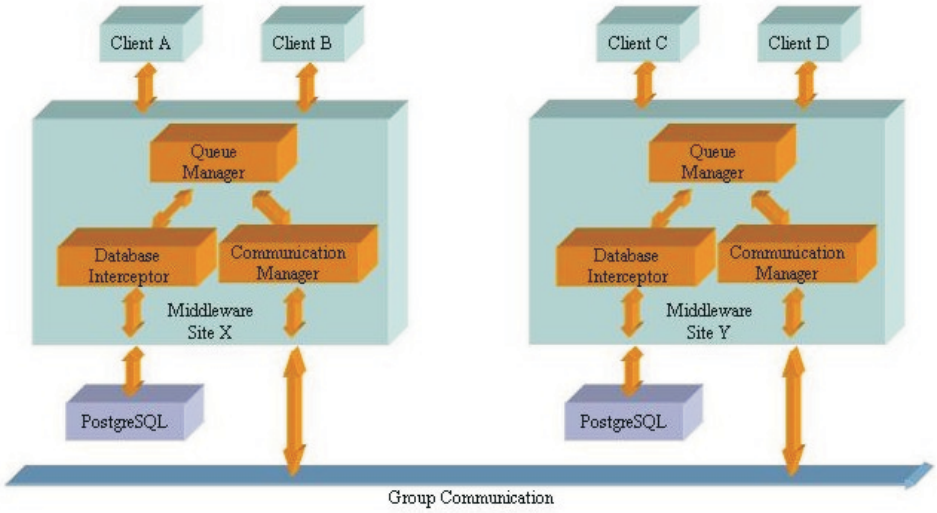


Fig. 9. Main components

The *queue manager* implements the core of the replication protocols. It maintains the conflict class queues and controls the execution of the transactions. It coordinates with the other sites and interacts with clients through the communication manager, and it submits transactions to the database through the database interceptor.

The *communication manager* is the interface between the queue manager and the group communication system (in our current implementation Ensemble [Hayden 1998]). The communication manager establishes the connection with the group, and monitors the group

membership. Sites that are not in the primary partition are forced to suicide. The communication manager also pipelines all messages into the network, and enforces the atomicity needed to execute actions associated to communication events, such as optimistic and definitive delivery of transaction and commit and view changes, by sequentially delivering such events.

The queue manager interacts with the database through the *database interceptor*. The database interceptor executes the application programs and controls the database transactions. In our current implementation, we use PostgreSQL [PostgreSQL 1998], version 6.4.2, as underlying database system. The database interceptor keeps a pool of processes available, each one of them with an open connection to the database. This acts as the connection pooling mechanisms found in modern middleware tools. Using this pool, transactions can be submitted and executed without having to pay the price of establishing a connection for each of them. At the same time, the processes can be used concurrently, thereby allowing the queue manager to submit to the database several transactions at the same time (if they don't conflict). The queue manager also limits the maximum number of open connections preventing a degradation of the underlying database in a sudden request burst (PostgreSQL allows to open more connections that it can handle yielding to a degradation of the service during peak loads).

7.2 Execution of Update Transactions

In what follows, we describe the implementation of NODO. The implementation of REORDERING is similar except for the fact that some transactions are reordered to prevent aborts.

Upon receiving a request from a client, the queue manager checks whether the requested transaction (application program) is a query or an update transaction. If it is a query, then it is executed locally. Otherwise, the request is TO-multicast to all sites.

When the communication manager OPT-delivers an update transaction T, it forwards the transaction to the queue manager. The queue manager inserts the transaction into the queues of all the basic classes included in C_T. If T is a local transaction and at the head of all of its queues, the queue manager sends the transaction to the database interceptor. The database interceptor issues a begin transaction (BOT), and executes T submitting SQL statements to the database. Once execution is completed, the database interceptor informs the queue manager (without committing the database transaction). The queue manager will not submit T's commit until its definitive order is confirmed (the transaction is TO-delivered). If the definitive (TO-delivery) and tentative order (OPT-delivery) agree, the queue manager will request T's write set through the database interceptor. The queue manager will send a commit message with the updates to all the sites through the communication manager. Once the message is delivered locally, the queue manager submits T's commit to the database interceptor, and removes T from the queues. The database interceptor commits T at the database. If the definitive (TO-delivery) and tentative order (OPT-delivery) do not agree, the queue manager establishes whether the ordering problem affects transactions that conflict. If this is not the case, the ordering problem is ignored (the transactions do not conflict; transitive closures for transactions accessing compound classes that overlap are also captured by the TO-delivery). If the ordering mismatch affects transactions that conflict, the serialization order obtained so far (following the OPT-delivery order) is incorrect (it should have followed the TO-delivery order) and T must be aborted. This is done by issuing an abort to the database interceptor and reordering T accordingly

in the queues. No communication with the rest of the sites is needed, since they will not apply the transaction updates until the master site completes.

If the transaction is remote, the queue manager waits until the transaction is at the beginning of all its queues, it is TO-delivered, and its commit message has been delivered. Then, the queue manager submits the write set to the database through the database interceptor. We will see in Section 7.5 how the write set is generated and how it is applied to the database. Once the transaction is completed, the queue manager will remove it from all the queues.

Several transactions can be submitted concurrently to the system. If they do not conflict they will be executed in parallel. Correct serialization is guaranteed as a combination of the following mechanisms. The communication manager delivers messages serially to the queue manager. The queue manager processes the actions on queues upon a certain event atomically (e.g. upon OPT-delivery including a transaction in all its queues, upon TO-delivery performing the reordering actions, etc.). At the same time, parallelism is achieved since the database interceptor and its processes execute concurrently to the queue manager. The queue manager takes care that for each conflict class only at most one transaction accessing this conflict class is submitted to the database interceptor, but transaction accessing different conflict classes can be executed concurrently by the database interceptor.

7.3 Execution of Queries

Queries (read only transactions) are executed only at the site they are submitted. Queries are executed using snapshot isolation so that they do not interfere with updates. However, if the database system does not provide snapshot isolation, the system gives queries a preferential treatment. In this case, queries are sent to a single site, as long as this site makes sure that the query does not reverse the serialization order of update transactions, it can execute the query at any time. This can be easily enforced by queuing the query after transactions that have been TO-delivered and before transactions that have not yet been TO-delivered. By doing this, the site can be sure that, no matter what happens to the update transactions, their serialization order will not be altered. This shortcut allows us to simulate snapshot isolation in a straightforward way and can be used with any other database management system that does not provide snapshot isolation (e.g., object oriented databases). It has the advantage that queries do not affect updates and do not need to limit their access to a single compound conflict class. This is an interesting option for monitoring and analysis purposes since it allows queries to be run on a single site without any restrictions on the items they can access. The proposed approach guarantees full serializability of queries since a query Q sees the state of the database as of the time all transactions TO-delivered before Q have committed and all transactions TO-delivered after Q have not yet started. The disadvantage is that while queries are executed on a conflict class, update transactions are delayed. This is the same problem as traditional strict 2-phase-locking has [Bernstein et al. 1987]. If the application is willing to have a lower level of isolation than serializability for queries, the queue manager can submit queries to the database interceptor without inserting them in queues. The database interceptor submits them to the database immediately using a low isolation level (e.g. read committed) in which read only transactions do not keep locks until the end, and hence, might read non-serializable data.

7.4 Optimistic Delivery

We have used Ensemble [Hayden 1998] as communication layer. Ensemble is a group communication protocol stack providing virtual synchronous multicast with different reliability and ordering properties.

In our replication protocol, update transactions use the total order multicast with optimistic delivery. However, such a primitive is not available in any existing group communication system, including Ensemble. One way to implement this primitive is to modify the group communication protocol stack. In the current prototype we have implemented such a primitive on top of Ensemble. Although integrating the optimistic delivery into the protocol stack would have been more efficient, implementing it on top has served our purposes well.

In our implementation, each total order multicast is performed in two steps. First, the sender sends the message using IP-multicast. Immediately after that, the message is sent again using the Ensemble reliable total order multicast. The delivery of the first message represents the OPT-delivery, the delivery of the second one represents the TO-delivery. The IP-multicast message will either be received before the TO-multicast message or not received at all (i.e., lost due to buffer overrun). In the latter case, before delivering the total order message (TO-delivery), an OPT-delivery is automatically triggered. Note that the IP-multicast cannot be delivered after the TO-multicast because both are sent physically one after the other over the same Ethernet segment and there are no retransmissions of IP-multicast messages (they are UDP messages).

7.5 Interaction between Middleware and Database System

The replication protocol in our implementation resides outside the database. In fact, the management of conflict class queues (inserting/removing/submitting for execution) can be considered a special form of a 2-Phase-Locking (2PL) [Bernstein et al. 1987]. In 2PL there is queue per data item. When a transaction wants to access a data item an entry is appended to the queue. The first entry is considered a granted lock and the transaction holding the lock can access the data item. The others are waiting until the first entry is removed which will be done only when the transaction holding the lock terminates. What is different in our solution is that a transaction T is inserted in all conflict class queues it wants to access at the begin of T (same as inserting locks in a lock table for all objects T wants to access at the begin of T), and that reordering might take place. As such, our queue manager basically replaces the concurrency control method implemented in the database system. In fact, we only rely on the locking mechanism within the database for some special situations (e.g., when an abort occurs).

In terms of direct interfaces to the database engine, our implementation requires two services from the API of the database engine. The first is a service to obtain the write set of a transaction (the new physical values of the modified tuples) used by the master site. The second is a service that takes the write set as input and applies it (used by the remote sites). With this, remote sites do not need to reexecute the original SQL statements of a transaction but only directly access and modify the tuples to be changed. These two services are similar to the get update and set update methods available in FT-CORBA. In fact, they exist in most commercial databases although they are not always directly accessible (for example, similar services are used internally to write the redo log during normal processing and apply the redo log at recovery after crash). We extended PostgreSQL to support

these services at the API (as functions, the way PostgreSQL might be extended). If such services are not directly available it is still possible to use other mechanisms to emulate these services. For instance, although this might lead to higher overhead, it is possible to use triggers at master sites to track updates performed by transactions; these updates are then recorded as simple update SQL statements only on the records to be changed, and then applied at the remote sites. Thus, for all intents and purposes, the middleware we propose can be used with most commercial database engines.

7.6 Interaction between the Replication Middleware and the Client

The client sees as its interface application programs. Invocations consist of the identifier of the application program and its arguments. This API takes care of connecting to the replication middleware and interacting with it. This guarantees transparent access to the replicated database from existing applications.

Application programs are written by application programmers that do not need to know about the replication middleware. The definition of an application program consists of its signature (name, arguments and its type), the SQL statements, a boolean that determines the kind of transaction (query or update), and a conflict class function. This function determines the conflict classes the application program accesses based on its parameters and SQL statements. The middleware keeps the definition of the programs and the conflict class definition in a table that is loaded at initialization time.

In principle, the middleware could also provide a standard database access layer like ODBC or JDBC. Those layers allow the use of replication the middleware on top of databases from different vendors.

The definition of conflict classes is very flexible. One option, for transactions which usually access only one table, consists in defining basic conflict classes as tables. Another option is to define conflict classes with respect to attributes whose values partition the data more or less evenly. For instance, let us consider a typical e-book-seller application. In this application the book category is a good candidate for partitioning. Since customers usually buy books from the same category, most of the times transactions will fall into a single conflict class. For those cases in which the buyer buys books across different categories the transactions will span through a compound conflict class. This is in fact the optimal scenario for our middleware, that is, most transactions access a single partition and some of them access multiple partitions.

8. EXPERIMENTAL RESULTS

In this section we analyze the scalability and overall performance of the algorithms and the implementation we propose. It is important to emphasize that the absolute values of the results are only meaningful to a certain degree. They could be improved by simply using faster machines or by using a database other than PostgreSQL. The important aspect of these results is the trends they show in terms of behavior as the number of sites and the load in the system increases. We expect the relative behavior to be similar for higher throughput ranges.

8.1 Experimental Setup

All the experiments have been run in a cluster of 15 SUN Ultra-5_10 (440MHz UltraSPARC-IIi CPU, 2 MB cache, 256 MB main memory, 9GB IDE disk) connected through a 100Mbits

Fast Ethernet network. PostgreSQL, version 6.4.2, has a rather inefficient buffer management policy. To prevent PostgreSQL itself from becoming the bottleneck before we can measure the limits of the replication protocol, we use the no-flush option offered by PostgreSQL (nothing is flushed to disk at the end of the transactions).

The database used for the experiments consists of 10 tables, each with 10,000 tuples. Each table has five attributes: two integers, t-id (which also acts as the primary key) and attr1, one 50 character string (attr2), one float (attr3) and one date (attr4). The only index that is maintained is an index on the primary key. The overall tuple size is slightly over 100 bytes, which yields a database size of more than 10 MB. We did not consider larger databases since this will only reduce the conflict workload and, again, turn PostgreSQL into the bottleneck.

The number of basic conflict classes varies for different experiments. The masters of these basic conflict classes are evenly distributed among the sites. The compound conflict class of a transaction is hashed and the resulting hash number is used to select one of the masters of the basic conflict classes as master of the compound conflict class. It should be noticed that it is not required that the master site of a compound conflict class is also the master of one of the constituent basic conflict classes (this is just an optimization to increase the chances of successful reordering). Clients are evenly distributed among the machines and submit their requests to the local middleware layer with an exponentially distributed submission interval. The transaction workload is chosen such that each server has the same likelihood of being the master site of a transaction. The workload in the database is divided among update transactions and queries. Since there is an infinite range of possibilities in terms of how many read and write operations a transaction can have, we have simplified the workload to make the results better understandable. We will consider update transactions that do not perform any read operation (worst case). The percentage variation between read and writes in the workload is controlled by varying the relative number of update transactions vs. queries.

The structure of the transactions used in the experiments is as follows. Update transactions have one or more update operations of the type:

```
UPDATE table-i SET attr1="randomtext", attr2=attr2+4
  WHERE t-id=random(1-1000)
```

Queries are structured as operations that scan a whole table and perform aggregation operations over all the data they read:

```
SELECT AVG(attr3), SUM(attr3) FROM table-i
```

In most cases, update transactions perform 8 update operations and have been designed to take about the same time as a query (to ease the comparison). The transactions do not perform any other computation or application dependent tasks (e.g., generating web pages with results). Using more “real life” transactions will improve the performance of the system since such extra work is only performed on one site. However, since it is difficult to predict how long such extra work will really take, we have not included application dependent tasks in our experiments. As such, the transactions used in our experiment stress test the system, and serve to illustrate the overhead of the replication layer and the database system.

Parameters	Distr. Locking	Scale-out	Resp. Time	Communic.	Compound Classes	Aborts
# servers	1-15					5
Database size	10 tables of 10,000 tuples each					
Tuple size	approx. 100 bytes					
# conf. classes	16/90/900				30/90/900	
# CC per txn	1				1-3	
txn per second	10	max.	10-110	20-260	10-80	100
% update txn	100%	0-100%		100%		
# upd op. in txn	5	8		1	8	
# write set size	504	804		104	804	

Table II. Experiment Parameters

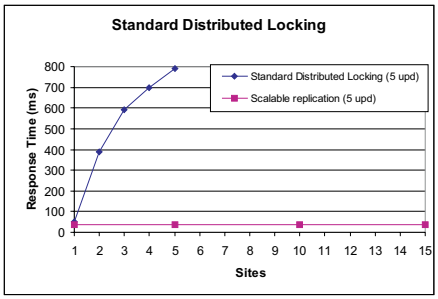
We have conducted 6 sets of experiments. The parameters used for each of these experiments are shown in Table II. In the first four experiments each transaction only accesses a single basic conflict class and the experiments focus on issues like scalability, response time behavior, and communication overhead. We have tested these experiments with 16, 90 and 900 conflict classes. 16 conflict classes represent a worst case scenario where – in order to have each transaction only access a single data partition – there exist only few, rather large conflict classes. A partitioning with 900 conflict classes represents a system where application processes access well defined, rather specific data. Experiment five analyzes the behavior of the system when transactions access more than one basic conflict class. These first five experiments use the REORDERING algorithm. The last experiment compares abort rates of NODO and REORDERING. Each test run submitted 2000 transactions to the system. The result of the first and last 10% of submitted transactions was ignored. The data points shown are calculated as the average of the remaining result values [5].

8.2 Comparison with traditional Distributed Locking

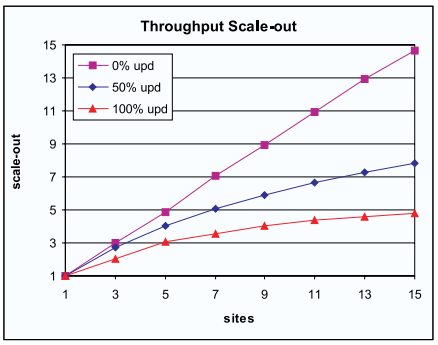
A first question that needs to be addressed is whether the middleware we propose really solves the limitations of conventional replication algorithms (e.g., those described in [Bernstein et al. 1987]). Gray et al. [Gray et al. 1996] showed that these conventional algorithms do not scale and, in particular, that increasing the number of replicas would increase the response time of update transactions and produce higher abort rates. We have compared the scalability in terms of response time of our solution with the standard distributed locking implementation of a commercial product, Oracle. Figure 10(a)) shows the response time of both approaches when the load is fixed to 10 update transactions per second and the number of sites increases from 1 to 5. The results for Oracle were taken from [Kemme and Alonso 2000a]. A slightly different computing environment was used (cluster of 5 PCs, 266 MHz, 128MB RAM, two 4GB disks and a switched full duplex 100MBit Fast Ethernet). Hence, only the form of the curves but not the absolute values should be considered.

The results for distributed locking reflect the behavior predicted by Gray et al. The

[5] Note, that it was not possible to use confidence interval calculations. PostgreSQL is a multiversion system, i.e., an update does not overwrite the old record but invalidates it and appends a new record. As such the number of records (invalid and valid) increases continuously throughout the test run. As a result, response times continuously increase (although only very slightly). Note that this also happens when using PostgreSQL in a centralized, not replicated setup. Hence, we performed our tests only when we had exclusive access to the cluster to avoid any disturbance. For each run, we only submitted a limited number of transactions, and made sure that there were no unexplainable outliers in the result set. We reinitialized the database upon each test run.



(a) Comparison with Distributed Locking



(b) Scalability of the system

Fig. 10. (a) Comparison with Distributed Locking and (b) Scalability

behavior clearly corresponds to a system that does not scale: for a fixed load, the response time increases as the number of sites increases. In this experiment the long response times are mainly due to the fact that distributed locking has a significant amount of messages all within the boundaries of the transaction. Our system, in comparison, was quite stable. For the range of sites explored, the response time did not vary, showing that the message overhead is not significant and that the system is easily able to handle the small extra *load* when more sites are added.

8.3 Throughput Scale-out

The main motivation for this work is to provide a replication algorithm that can scale in a cluster based system. Black box approaches have to execute all update transactions at all sites since they do not have any additional knowledge about the database system. As a result, adding new sites in an update intensive environment might help for fault-tolerance, but cannot be used to scale up the system. Using a grey box approach we hope to achieve both fault-tolerance and scalability.

Hence, this experiment analyzes how the throughput scales up when we increase the number of sites. We have run three sets of tests to see how the system behaves when the workload is read only, write only, and a mixture of both. Figure 10(b) shows the scale-out factor when we increase the number of sites in the system from 1 to 15. The scale-out factor for a given system size is calculated as the maximum throughput that can be achieved in this setting divided by the maximum throughput in a single-site system. The results shown are for a system with 16 basic conflict classes. For 90 or 900 conflict classes the results are similar.

Read only workloads (0% updates) have obviously optimal scalability since queries are executed completely locally. Hence, n sites can achieve an n-times higher throughput than a single site. At the other extreme, for a write-only workload (100% updates) one would expect that adding new sites does not increase the overall throughput (or might even decrease it) since every site has to apply all updates. Still, in our approach, 5 sites perform three times the throughput of a single-site system, and 15 sites have a scale-out close to 5. However, we can see that the scale-out is not linear and seems to reach a saturation point. The reason for the good performance at low system sizes is the use of asymmetric

processing [Jiménez-Peris et al. 2003] in which update transactions are executed by one site and the updated tuples propagated to the remaining sites. We have discussed the availability of such functionality in Section 7.5. Since the application of updated tuples has a lower overhead than executing the original update SQL statement at all sites, and hence, we can achieve scale-out factors greater than 1. Still, the overhead is not negligible. Eventually, adding new sites does not result in an increase in throughput. The results for a mixed workload (50% updates) indicate how real life applications will perform: depending on the percentage of queries, the scale-out will lie between the results of the two extreme configurations.

8.4 Response Time Analysis

This experiment looks at the response time behavior, and determines at which throughput the system saturates (i.e., response times deteriorate). As in the previous experiment, we consider transaction workloads of 0%, 50%, and 100% updates. We conducted test suites with 5, 10, and 15 sites. Within each test suite, we increased the load until the system saturated. The results are shown in Figure 11 using two different representations. In each of the Figures 11(a-c), the number of sites is fixed and the curves show the three different transaction workloads. In Figures 11(d-f), the transaction workload is fixed and the three curves show different system sizes.

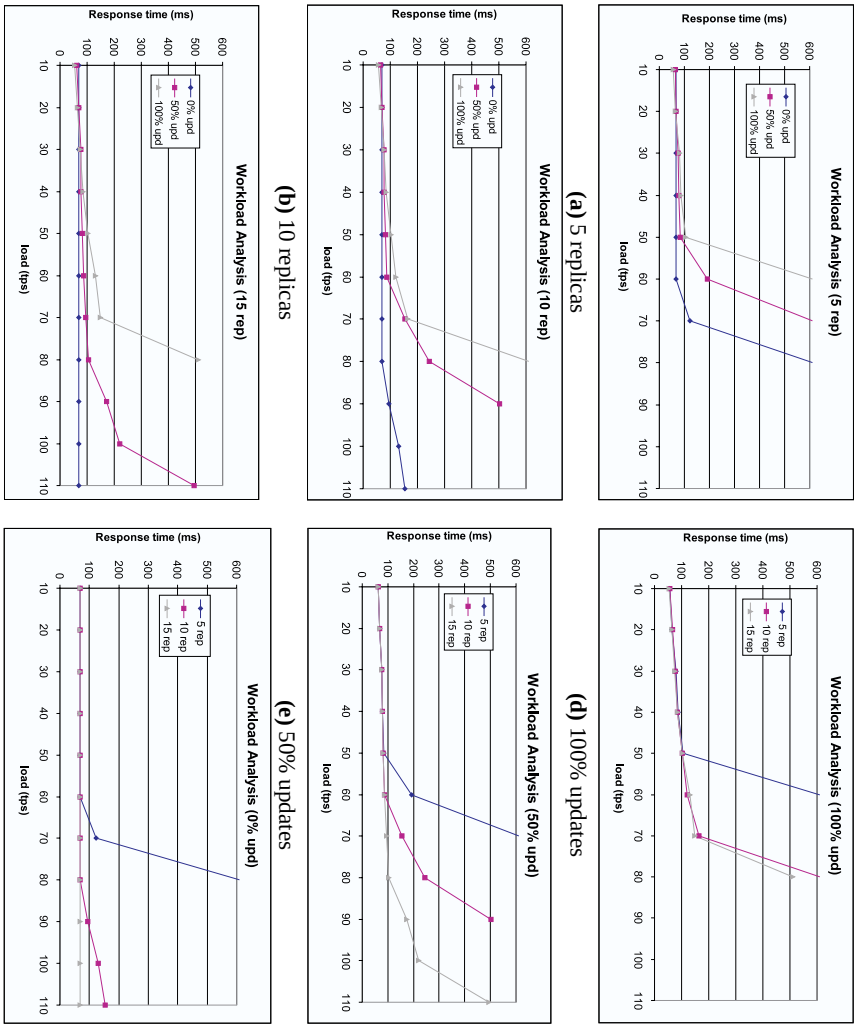
In all figures, we can observe a relatively flat evolution of the response time until the system is saturated and can no longer respond. The response time is nearly independent of the transaction workload as can be best seen in Figures 11(a-c): at low transaction rates, there is enough processing power to process write sets without affecting the response times of the individual transactions. As the load increases, the spare time diminishes and thus, the time devoted to process write sets starts to lightly affect the other transactions. Interestingly, the response time is also independent of the number of sites in the system (best seen in Figures 11(d-f)). This is true because in all configurations only two messages per transaction are sent. In addition, the growth in response time when the saturation point is reached is less explosive as the number of sites increases. This indicates that larger systems have a more graceful degradation than smaller ones.

Regarding the saturation point, Figures 11(a-c) show that the higher the proportion of read transactions the more transactions the system can process before becoming saturated. Figures 11(d-f) indicate that the more sites are in the system the more transactions the system can handle. And this, without a negative effect on response times. However, the performance of any replication algorithm is strongly determined by the proportion of updates in the workload, and adding new sites when there are 100% updates will only work up to limited system sizes (see Figure 11(d)). This reflects the scalability results of the previous experiment.

8.5 Communication Overhead

When using group communication primitives, the system built can only scale as much as the underlying communication tool. One of the typical problems of conventional replication algorithms is that they easily overload the network by generating too many messages (e.g., distributed locking generates one message per operation per transaction per site; a 10 site system running transactions of 10 operations at 50 transactions per second generates 5,000 messages per second).

In particular, our implementation requires two messages per transaction and it could be



(a) 5 replicas **(b)** 10 replicas **(c)** 15 replicas **(d)** 100% updates **(e)** 50% updates **(f)** 0% updates

Fig. 11. Response time for different transaction workloads and configurations

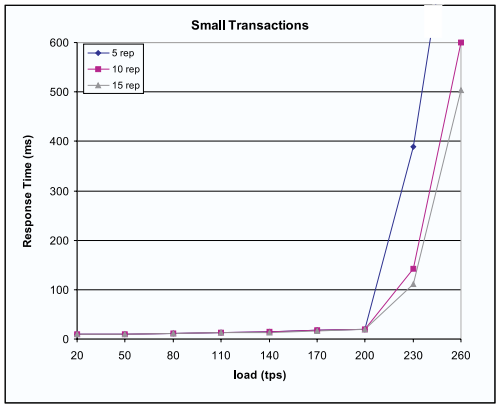


Fig. 12. Analyzing the communication overhead using small transactions

questionable whether the optimistic algorithm we use is feasible in practice. In order to test this aspect of the system, we have performed a test with very small update transactions (containing a single update) in order to execute as many transactions as possible and hence, produce many messages. The response time was measured for increasing loads and different configurations until the system was saturated. Fig. 12 shows a flat response time up to quite high transaction loads (over 200 transactions per second). This indicates that the communication does not become a bottleneck up to that point where the system saturates. At that stage, it does not matter what happens to the communication layer since the system is incapable of dealing with the load anyway. Thus, for the purposes of cluster based systems, the use of group communication primitives does not seem to be the limiting factor.

A last point to note regarding this experiment is the difference in scalability for short transactions (Fig. 12) and medium transactions (Fig. 11(d)). The reason is that for short transactions the constant overhead associated with processing a transaction remotely (start-up, commit) is quite large in relative terms. Thus, there is not that much redundant work to reduce. The longer the transaction the bigger the effect of reducing the writing overhead and the higher the scalability.

8.6 Compound Conflict Classes

Our implementation of conflict classes at the middleware layer is a simplified version of the concurrency control mechanism in the database, and conflict classes are typically coarser than the locking granularity within the database. The reason for coarser granularity is that the algorithms require knowing all conflict classes to be accessed at the beginning of the transaction. This is often not possible on a fine level. We believe this might be the main performance disadvantage of the grey box approach over a white box approach that implements replication within the database, and hence, can take advantage of the concurrency control mechanisms within the database.

Hence, in this section we analyze transactions that have varying conflict behavior. In particular, we consider transactions that access compound conflict classes (i.e., more than one basic conflict class) and we consider systems with a different total number of basic

conflict classes. By introducing compound conflict classes, it is easier to split the database into smaller partitions since we do not have the restriction that each transaction may only access one partition. With this, we have a higher potential of load balancing and concurrency. On the other hand, given a fixed number of conflict classes, when transactions access several conflict classes, transactions have a higher chance of conflict which might lead to data contention and possibly suboptimal use of resources.

The first test analyzes how conflict class configurations influence conflict rates, and as a result, the response times of transactions. Conflict classes influence conflict rates in two ways: (1) the more conflict classes a transaction accesses, and (2) the smaller the total number of conflict classes, the higher the conflict rate. Our test suite investigates the impact of both of these dimensions.

Figure 13 shows the response time for an increasing load in the following configurations: in column (a) the system has 30, in (b) 90, and in (c) 900 conflict classes. In each of the columns, transactions access a single conflict class in the top figure, two conflict classes in the middle figure, and three conflict classes in the bottom figure. The workload consists exclusively of update transactions.

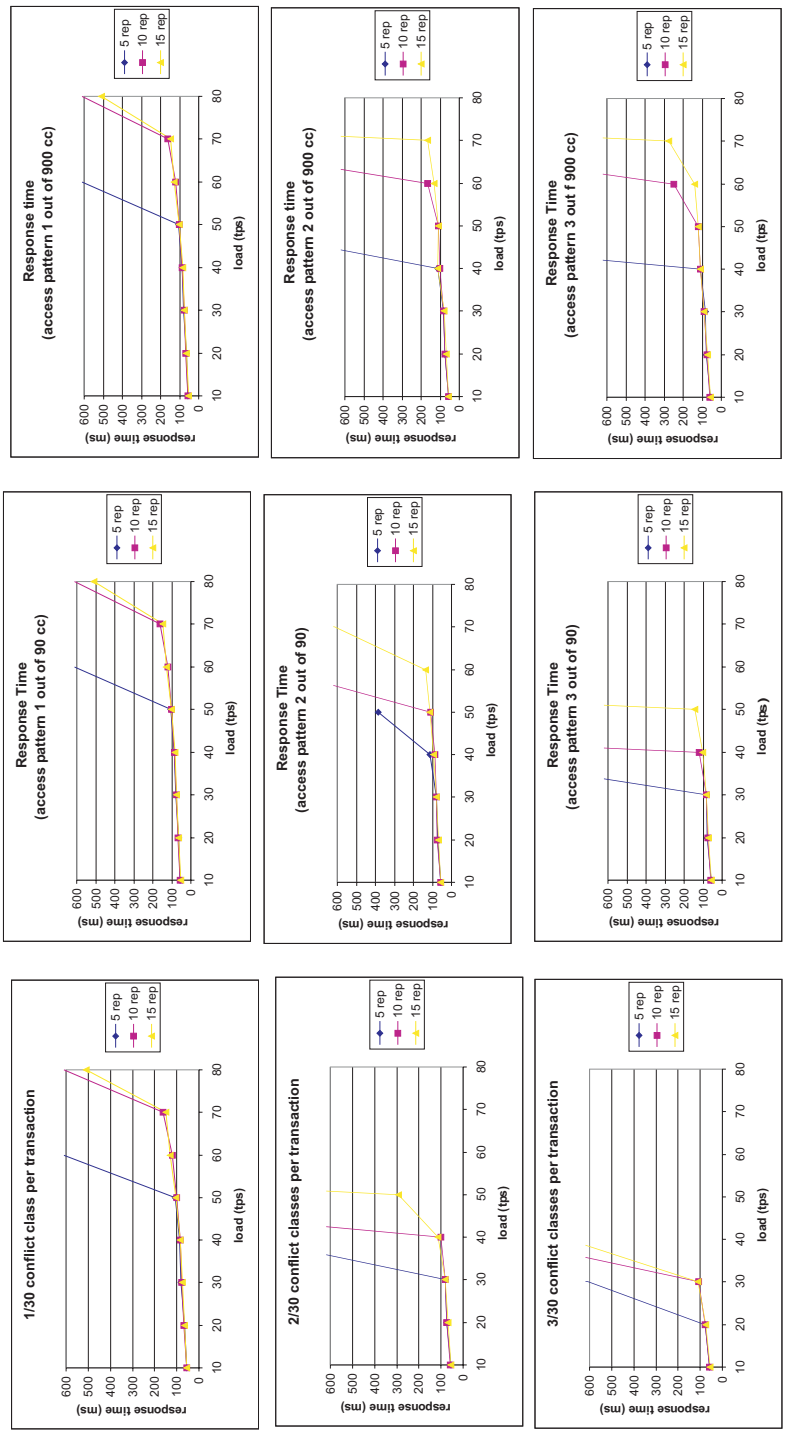
Looking at column (a) with only 30 conflict classes in the system, we can observe that transactions with two or three conflict classes have a considerably smaller maximum throughput than transactions accessing only one conflict class. This is due to the very high conflict rates in such a scenario. For a load of 50 tps, there are on average 5 transactions concurrently in the system. If each of them accesses 2 conflict classes, they access in total 10 conflict classes. With this, the probability that at least two transactions access the same class is 80% ($1 - \Pi_{i=0}^{9} \frac{30-i}{30}$), if transactions access 3 conflict classes, we can expect at least one conflict with 99% chance. Such conflicts might occur between transactions executed at different sites (although conflicting in one basic conflict class they still might access different compound classes that have different masters). As a result, if transaction T_i executed at site N_i conflicts with transaction T_j executed at N_j, and T_i is ordered before T_j, then N_j must wait and is possibly idle (if it only masters C_{T_j}) until T_i is executed. Hence, in such a high conflict scenario we cannot take full advantage of the enhanced processing power. Still, some degree of scalability is possible.

By introducing more conflict classes (columns (b) and (c)), we can alleviate this problem. With 90 classes, performance loss is already less than for 30 conflict classes, and with 900 conflict classes we have nearly eliminated the problem.

Another fact that can be observed in the graphs is that whilst the saturation point is not reached, for a given load the response time is nearly the same independently of the total number of conflict classes and the number of conflict classes accessed by a transaction.

In regard to scalability, Figure 14 shows that the scale-out factor is only slightly affected, i.e., the scale-out for transactions accessing more than one basic conflict classes is only slightly smaller than for transactions with one basic conflict class. The figure shows the results for 900 conflict classes, but the results are similar for 30 and 90 conflict classes. This means, that although transactions accessing several conflict classes achieve smaller throughputs than transactions with one conflict class for a given system size, adding new sites will increase the maximum throughput in all configurations.

We have not directly compared the approach with a white box approach. However, the performance results in [Kemme and Alonso 2000a], showing a white box approach also using PostgreSQL and Ensemble in a similar computing environment, indicate that in case



(a) 30 Conflict classes **(b)** 90 Conflict classes **(c)** 900 Conflict classes

Fig. 13. Response time for different transaction workloads and configurations

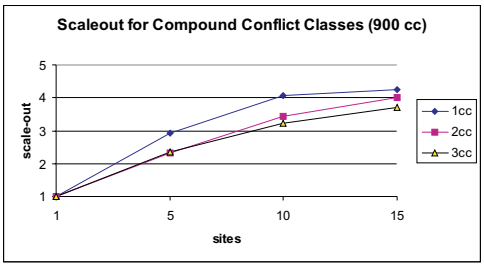


Fig. 14. Scalability of Compound Conflict Classes

we are able to perform smart data partitioning, the middleware based grey approach does not perform significantly worse than the white box approach.

The main result of this experiment is that in order to achieve efficient concurrency control in the middleware, we must be able to perform a smart data partitioning. A first acceptable solution is to partition the data into coarse data partitions in such a way that most transactions only access one partition while only few span several conflict classes. Then, we only have to distribute the partitions such that each site has more or less the same load. Conflicts between sites will be very rare. An alternative solution, whenever possible, is to split data into small partitions, yielding to a high number of conflict classes. Hence, it will not matter anymore how many conflict classes are accessed by each transaction.

8.7 Aborts

In order to keep one-copy serializability, replica control algorithms abort transactions. In the first four experiments the abort rate was virtually null. In the experiment of the previous section the abort rate was never higher than 0.2%. The reason was mismatches between the optimistic and definitive order of the TO-multicast were very rare. This was the case due to we used a local area network, and neither message load nor message size were very big. Also, the total order protocol used by Ensemble is sequencer-based. That is, it uses as total order the order in which messages arrive at a specific site, the sequencer, and hence, the messages received by the sequencer are broadcast on the network (and received by most sites).

In this experiment we emulate an environment where mismatches occur more often. This will be the case in extended or wide area networks, or if using a different total order protocol than the one implemented by Ensemble is used. The NODO algorithm aborts a local transaction when there is a mismatch between the optimistic and definitive order for conflicting transactions. The REORDERING algorithm avoids some of these aborts. Depending on the degree of mismatch, we study the proportion of aborted transactions induced by NODO and REORDERING, and more concretely the amount of aborts prevented by REORDERING with respect to NODO.

In the following we quantify the mismatch between the optimistic and definitive delivery orders as the *degree of disorder*. In order to enable the comparison of the two algorithms for different degrees of disorder, we have devised an experiment in which the degree of disorder is generated in a synthetic way. In this way it becomes possible to observe the behavior of aborts for increasing degrees of disorder. The degree of disorder is determined by two parameters, disorder rate and disorder distance, and it is modeled in the following way.

1 CC	2 CC	3 CC
100 %	0%	0 %
50 %	50 %	0 %
80 %	20 %	0 %
90 %	10 %	0 %
50 %	40 %	10 %
80 %	15 %	5 %

Table III. Composition of the workload in terms of percentage of transactions accessing 1 to 3 conflict classes

1 CC	2 CC	3 CC	% Aborts saved 30 CC	% Aborts saved 90 CC
100 %	0 %	0 %	100	100
50 %	50 %	0 %	25-45	30-50
80 %	20 %	0 %	65-80	60-70
90 %	10 %	0 %	70-95	85-100
50 %	40 %	10 %	20-35	20-25
80 %	15 %	5 %	60-80	55-65

Table IV. Aborts saved by REORDERING with respect to NODO for the different workloads

First, the sequence of messages to be submitted is built. Then, messages are exchanged randomly as many times as indicated by the disorder rate. For instance, in a sequence of 1000 messages, a 10% disorder rate means that 100 messages picked randomly are exchanged. The disorder distance determines the distance between the messages to be exchanged (e.g., 1 means a message is exchanged with its direct neighbor). In our experiments, we fixed the distance to be 3.

The experiments use different workloads in terms of number of conflict classes accessed by each transaction. A transaction can access up to 3 basic conflict classes. Table III shows the tested workloads. The first workload consists of 100% of transactions accessing only one basic conflict class. The second workload consists of 50% transactions accessing one basic conflict class and 50% transactions accessing two basic conflict classes, and so on. In the last workload, an 80% of the transactions access a single basic conflict class, a 15% access two conflict classes, and a 5% access three conflict classes. We conducted experiments with a total number of 30, 90 and 900 basic conflict classes.

Fig.15.a-c shows the percentage of aborts in the different setups for NODO and Fig.15.d-f for REORDERING. From left to right, the number of conflict classes increases. The individual graphs in each figure show the different workloads. When looking at the figures from left to right we see that the higher the number of conflict classes, the lower the conflict rate, and hence, the smaller the number of aborts. When looking at the different graphs within a figure we can observe that the more transactions access only one conflict class, the smaller the number of aborts. If we compare the same settings for NODO and REORDERING, we see that REORDERING exhibits much smaller abort rates than NODO.

Table IV summarizes the improvement in the abort rate of REORDERING with respect to NODO [6]. The rates of prevented aborts range from 20-100%. The 100% saving only happens when the workload is composed exclusively of transactions accessing a basic conflict class. In this case, it is always possible to apply reordering. The highest rates

[6] Results for 900 CC not included because they were either 0% or 100% due to the low number of aborts.

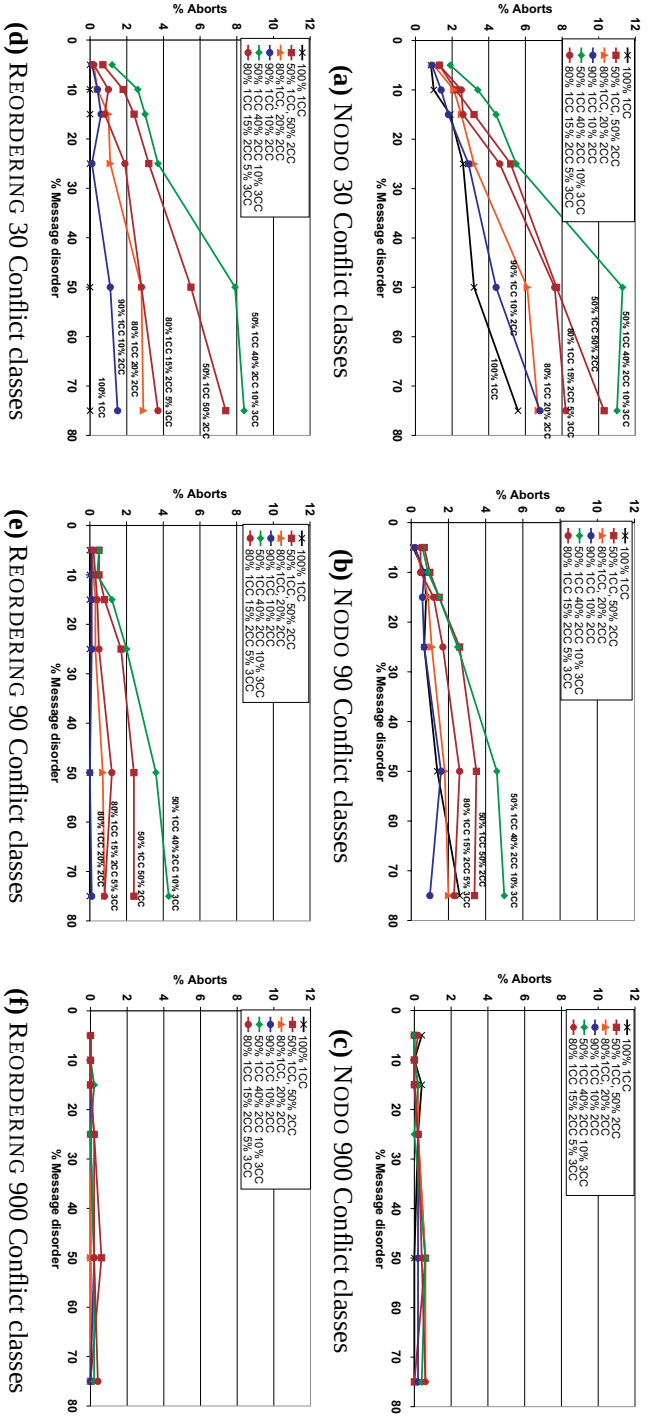


Fig. 15. Abort rates in NODO and REORDERING

of prevented aborts (above 60%) are exhibited when at least 80% of transactions access a single basic conflict class. The lowest rates of saved aborts occur when the fraction of transactions accessing a single basic conflict class is only at 50%. The rational behind these results is that transactions accessing only one basic conflict class can be reordered in regard to other transactions only accessing this class or to transactions with a larger number of conflict classes. In contrast, the probability of being able to reorder a transaction accessing more than one basic conflict class is lower.

Finally, we would like to highlight that in the case a single site observes a high abort rate due to message disorder, it is quite simple to deactivate the optimistic transaction processing by delaying the actions upon OPT-delivery to the time of TO-delivery. If all sites observe a high degree of disorder, OPT-delivery can be completely deactivated in an adaptive fashion. This would eliminate the aborts in those scenarios in which the abort rate increases beyond an acceptable threshold.

9. RELATED WORK

Conventional database replication protocols are well known and their correctness is studied in much detail [Bernstein et al. 1987]. Being of an eager nature (coordination among the replicas takes place before transaction commit) they have been mainly designed for fault-tolerance. Unfortunately, these protocols suffer from severe limitations that render them impractical [Gray et al. 1996]. As a result, commercial systems mainly use lazy approaches (updates are only propagated after the transaction commits) favoring performance over fault-tolerance and correctness.

Over the last years several research groups have attempted to address these limitations by using a combination of database techniques and group communication primitives. The basic idea is to use total order message delivery to guarantee that all replicas see a consistent order of the transactions to execute. As long as a replica produces a serialization order that does not contradict the imposed total order, consistency is guaranteed across the system. In most of the proposed algorithms, an eager propagation mechanism guarantees fault-tolerance. Several protocols have been proposed along these lines, exploiting different degrees of isolation [Kemme and Alonso 2000b], analyzing the problems of conventional replication protocols [Stanoi et al. 1998; Holliday et al. 1999; 2000], and exploring alternative protocol designs [Pedone and Frolund 2000]. [Pacitti and Simon 2000] combines the total order concept with a lazy replication strategy. In [Pedone et al. 1997] an optimistic replication protocol is presented. This protocol presents a complementary approach for transaction reordering to the one presented in this paper. The protocol of [Pedone et al. 1997] is aimed to reduce aborts in an optimistic concurrency control protocol by delaying transaction certification, what results in an increase of response times. This contrasts with the REORDERING algorithm in which transaction processing is not delayed but performed in advance, therefore reducing response times.

In terms of implemented systems using group communication primitives, Postgres-R is a replicated database implemented as an extension of PostgreSQL [Kemme and Alonso 2000a]. In Postgres-R, replication is implemented within the database engine (i.e., take a white box approach). This allows a close and very efficient interaction between communication and transaction management and includes important optimizations that minimize the overhead of replication. Transactions are processed at a single site, and updates are forwarded to the other sites. A similar approach is also used in commercial replicated

databases based on lazy replication protocols. [Rodrigues et al. 2002] implement a similar replication strategy within an object oriented database. Outside the database, [Amir and Tutu 2002; Amir et al. 2002] implement replication at the middleware layer using a black-box approach that has been tested both in a LAN and in a WAN.

There is an intrinsic tradeoff between black and grey box approaches. The principal advantage of the black box approach is that it does not require any service from the database. This solution can scale by introducing queries in the workload. The scalability is more limited than in the grey box approach, since all sites fully process all update transactions [Jiménez-Peris et al. 2003]. On the other hand, the grey box approach requires some services from the database and some knowledge about the application semantics (the data partitions) thereby reducing the generality of the solution. In return, it enhances scalability even for workloads with a high percentage of write operations [Jiménez-Peris et al. 2003].

Since the proposal of conflict-aware transaction scheduling at the middleware level [Patiño-Martínez et al. 2000; Jiménez-Peris et al. 2002], new projects have built on this approach for different purposes, for instance, to improve the performance of web dynamic contents [Amza et al. 2003] and to build clustered JDBC drivers [Kistijantoro et al. 2003].

Wool [Wool 1998] argues that quorums may become an alternate to the read-one write-all (ROWA) approach for database replication. Wool argues that reading locally might not be faster than on another site due to the increasing speed of the networks compared to disks, and some quorum systems incur in less work than ROWA even for a low proportion of update transactions in the workload. This result holds for symmetric systems, where all sites do the same work for each transaction. However, our system is asymmetric, a site executes the whole update transaction, while the rest just apply the resulting updates. [Jiménez-Peris et al. 2003] shows that asymmetric processing favors ROWA. Also, when considering database systems, it is not clear how to perform version comparison needed to read the latest version of a data item if complex SQL query statements are involved.

The recovery of failed replicas is a complex issue that is described elsewhere. In [Amir and Tutu 2002], it is discussed how to guarantee the consistency of a database in the advent of partitioning and recovery of replicas by making use of uniform multicast [Chockler et al. 2001] and extended virtual synchrony [Moser et al. 1996]. Online recovery has also been studied in two different contexts: in protocols implemented within the database [Kemme et al. 2001], and at the middleware level [Jiménez-Peris et al. 2002]. In particular, the latter focuses on a provably correct online recovery algorithm for the DISCOR algorithm.

Another area where a lot of attention has been devoted to replication is fault tolerant object containers, and more concretely, FT-CORBA [OMG 2000]. Eternal [Narasimhan et al. 2002] follows a black box approach to replication of CORBA servers. This system supports active and passive replication for different multithreading models. Consistency is achieved by using the group communication system Totem [Agarwal et al. 1998]. On the other extreme of the spectrum, Electra [Maffeis 1995] provides fault-tolerance following a white box approach. In this work a CORBA implementation is modified and enhanced with group communication to implement replication. A different approach was taken in OGS [Felber et al. 1998] by providing replication through a set of CORBA services. This approach is more oriented towards providing facilities to build fault-tolerant applications contrasting with the other approaches where transparent fault-tolerance was the main goal. A similar approach providing services for fault-tolerance in CORBA is described in [Morgan et al. 1999]. The Friends system [Fabre and Perennou 1998] uses a reflective approach

to add fault tolerance at the application level. This approach is based on a combination of metaobjects, group communication and security. Some others approaches for fault-tolerant CORBA include AQuA [Ren et al. 2003] and DOORS [Natarajan et al. 2000].

Over the last years, there has also been a wide range of proposals for efficient and consistent database replication that is not based on group communication. [Chundi et al. 1996; Pacitti and Simon 2000; Breitbart et al. 1999] look at lazy replication solutions that guarantee serializability by restricting the placement of copies and controlling the order in which updates are applied at remote sites. Another approach combines eager and lazy replication techniques [Breitbart and Korth 1997; Anderson et al. 1998]. Sites coordinate before committing transactions to guarantee serializability but the updates are only sent and applied after the commit. All of these approaches follow a primary copy approach where updates to a certain object may only be performed by the primary copy. Transactions are not allowed to update primary copies residing on different sites. Our approach, although being primary copy, does not have this restriction.

10. CONCLUSIONS

Replication has become a key technique in web farms and database clusters. Unfortunately, there is a big gap between known replication protocols and practical solutions. In this paper, we have extended previous work by ourselves and other authors in order to provide a middleware tool that supports consistent and scalable data replication. The protocol implemented includes several important optimizations needed to achieve good performance. One of the contributions of the paper is precisely this: to combine the generality of a middleware based tool with the optimizations typical of solutions implemented at the database level. The experimental results presented prove the feasibility of the approach and indicate how the design parameters can be varied to tune the behavior of the system.

We are currently using this middleware platform to implement a large web farm and exploring a number of interesting features that can be combined with the replication protocol here presented. Some of these features include load balancing, on-line recovery, and autonomic behavior by increasing the number of replicas in the system in response to changes in the overall load.

REFERENCES

AGARWAL, D., MOSER, L., MELLIAR-SMITH, P. M., AND BUDHIA, R. 1998. The Totem Multiple-Ring Ordering and Topology Maintenance Protocol. *ACM Transactions on Computer Systems 16*, 2, 93–132.

AMIR, Y., DANILOV, C., MISKIN-AMIR, M., STANTON, J., AND TUTU, C. 2002. Practical Wide Area Database Replication. Tech. Rep. CNDS-2002-1, Johns Hopkins University.

AMIR, Y. AND TUTU, C. 2002. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*.

AMZA, C., COX, A. L., AND ZWAENEPOEL, W. 2003. Distributed Versioning Consistent Replication for Scaling Back-end Databases for Dynamic Content Web Sites. In *Proc. of Int. Middleware Conf. LNCS-2672*. Springer.

ANDERSON, T., BREITBART, Y., KORTH, H. F., AND WOOL, A. 1998. Replication, Consistency, and Practicality: Are These Mutually Exclusive? In *ACM SIGMOD Conference*.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA.

BERNSTEIN, P. A., SHIPMAN, D., AND ROTHNIE, J. B. 1980. Concurrency Control in a System for Distributed Databases (SDD-1). *ACM Trans. on Database Systems 5*, 1, 18–51.

BIRMAN, K. 1996. *Building Secure and Reliable Network Applications.* Prentice Hall, NJ.

BREITBART, Y., KOMONDOOR, R., RASTOGI, R., SESHADRI, S., AND SILBERSCHATZ, A. 1999. Update propagation protocols for replicated databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*. Philadephia, Pennsylvania.

BREITBART, Y. AND KORTH, H. F. 1997. Replication and Consistency: Being Lazy Helps Sometimes. In *Proc. of the Principles of Database Systems Conf.* 173–184.

CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. 2001. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys 33*, 4 (Dec.), 427–469.

CHUNDI, P., ROSENKRANTZ, D. J., AND RAVI, S. S. 1996. Deferred updates and data placement in distributed databases. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*. New Orleans, Louisiana.

FABRE, J. C. AND PERENNOU, T. 1998. A Metaobject Architecture for Fault-Tolerant Distributed Systems. *IEEE Transactions on Computer Systems 47*, 1, 78–95.

FELBER, P., GUERRAOUI, R., AND SCHIPER, A. 1998. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems 4*, 2, 93–105.

FRIEDMAN, R. AND VAN RENESSE, R. 1995. Strong and Weak Virtual Synchrony in Horus. Tech. Rep. TR95-1537, CS Dep., Cornell Univ.

GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. 1996. The Dangers of Replication and a Solution. In *Proc. of the SIGMOD*. Montreal, 173–182.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers.

HADZILACOS, V. AND TOUEG, S. 1993. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*. Addison Wesley, 97–145.

HAYDEN, M. 1998. The Ensemble System. Tech. Rep. TR-98-1662, Department of Computer Science. Cornell University. Jan.

HOLLIDAY, J., AGRAWAL, D., AND ABBADI, A. E. 1999. The Performance of Database Replication with Group Communication. In *29th Int. Symp. on Fault-tolerant Computing, Wisconsin*.

HOLLIDAY, J., AGRAWAL, D., AND ABBADI, A. E. 2000. Using Multicast Communication to Reduce Deadlock in Replicated Databases. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*. 196–205.

JIMÉNEZ-PERIS, R., PATIÑO-MARTÍNEZ, M., AND ALONSO, G. 2002. Non-Intrusive, Parallel Recovery of Replicated Data. In *IEEE Symp. on Reliable Distributed Systems (SRDS)*. Osaka, Japan.

JIMÉNEZ-PERIS, R., PATIÑO-MARTÍNEZ, M., ALONSO, G., AND KEMME, B. 2002. Scalable Database Replication Middleware. In *Proc. of 22nd IEEE Int. Conf. on Distributed Computing Systems, 2002*. Vienna, Austria.

JIMÉNEZ-PERIS, R., PATIÑO-MARTÍNEZ, M., ALONSO, G., AND KEMME, B. 2003. Are quorums an alternative for data replication? *ACM Transactions on Database Systems 28*, 3, 257–294.

KEMME, B. AND ALONSO, G. 2000a. 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.

KEMME, B. AND ALONSO, G. 2000b. A new approach to developing and implementing eager database replication protocols. *ACM TODS 25*, 3 (Sept.), 333–379.

KEMME, B., BARTOLI, A., AND BABAOGLU, O. 2001. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 2001)*. Goteborg, Sweden.

KEMME, B., PEDONE, F., ALONSO, G., SCHIPER, A., AND WIESMANN, M. 2003. Using Optimistic Atomic Broadcast in Transaction Processing Systems. *IEEE Trans. on Knowledge and Data Engineering 15*, 4.

KISTIJANTORO, A. I., MORGAN, G., SHRIVASTAVA, S. K., AND LITTLE, M. C. 2003. Component Replication in Distributed Systems: a Case Study using Enterprise Java Beans. In *IEEE Symp. on Reliable Distributed Systems (SRDS)*. Florence (Italy).

MAFFEIS, S. 1995. Adding Group Communication and Fault-Tolerance to CORBA. In *Proc. of 1995 USENIX Conf. on Object-Oriented Technologies*.

MORGAN, G., SHRIVASTAVA, S., EZHILCHELVAN, P., AND LITTLE, M. 1999. Design and Implementation of a CORBA Fault-tolerant Object Group Service. In *Proc. of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, DAIS'99*.

MOSER, L., MELLIAR-SMITH, P., AGARWAL, D., BUDHIA, R., AND LINGLEY-PAPADOPOULOS, C. 1996. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM 39*, 4 (Apr.), 54–63.

NARASIMHAN, P., MOSER, L. E., AND MELLIAR-SMITH, P. M. 2002. Eternal - a component-based framework for transparent fault-tolerant CORBA. *Software: Practice and Experience 32*, 8, 771–788.

NATARAJAN, B., GOKHALE, A., YAJNIK, S., AND SCHMIDT, D. C. 2000. DOORS: Towards high-performance fault-tolerant CORBA. In *Proc. of the Int. Symp. on Distributed Objects and Applications*.

OMG. 2000. *Fault Tolerant CORBA*. Object Management Group.

PACITTI, E. AND SIMON, E. 2000. Update Propagation Strategies to Improve Freshness in Lazy Master Replicated Databases. *VLDB Journal 8*, 3, 305–318.

PATIÑO-MARTÍNEZ, M., JIMÉNEZ-PERIS, R., KEMME, B., AND ALONSO, G. 2000. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*. Vol. LNCS 1914. 315–329.

PEDONE, F. AND FROLUND, S. 2000. Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases. In *Proc. of 19th Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, Nuremberg, Germany, 176–185.

PEDONE, F., GUERRAOUI, R., AND SCHIPER, A. 1997. Transaction Reordering in Replicated Databases. In *Proc. of 16th Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society Press, Durham, NC, 175–182.

PEDONE, F., GUERRAOUI, R., AND SCHIPER, A. 1998. Exploiting Atomic Broadcast in Replicated Databases. In *Proc. of 4th International Euro-Par Conference*, D. J. Pritchard and J. Reeve, Eds. Vol. LNCS 1470. Springer, 513–520.

POSTGRESQL. 1998. *v6.4.2*. http://www.postgresql.com.

REN, J., BAKKEN, D. E., COURTNEY, T., CUKIER, M., KARR, D. A., RUBEL, P., SABNIS, C., SANDERS, W. H., SCHANTZ, R. E., AND MOUNA, S. 2003. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. *IEEE Transactions on Computers 52*, 1, 31–50.

RODRIGUES, L., MIRANDA, H., ALMEIDA, R., MARTINS, J., AND VICENTE, P. 2002. Strong Replication in the GlobData Middleware. In *Proc. of the Int. Workshop on Middleware-Based Systems (part of DSN)*. G96–G104.

SKEEN, D. AND WRIGHT, D. D. 1984. Increasing the availability in Partitioned Database Systems. In *Proc. of the Principles of Database Systems Conf.* 290–299.

STANOI, I., AGRAWAL, D., AND EL ABBADI, A. 1998. Using broadcast primitives in replicated databases. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*. Amsterdam, The Netherlands.

WEIKUM, G. AND VOSSEN, G. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers.

WOOL, A. 1998. Quorum systems in replicated databases: Science or fiction? *Data Engineering Bulletin 21*, 4, 3–11.

A. 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 all protocols is the fact that transactions are enqueued (respectively rescheduled) in one atomic step. Hence, there is no interleaving between conflicting 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.

In the following, we denote for a given site, N, $T_1 \longrightarrow_N T_2$ if T_1 and T_2 conflict, and T1 was executed, and hence serialized, before T_2 at N. It denotes that all operations of T_1 appear before all operations of T_2 in N's history H_N.

A.1 Correctness of DISCOR

We will show that all sites serialize conflict transactions according to the order they are delivered at the master site. In here, we only look at sites that do not fail.

LEMMA 1 SERIALIZABILITY (DISCOR). *Let N be the master site of transactions T_1 and T_2. If T_1 was delivered at N before T_2 ($T_1 \longrightarrow_{DEL_N} T_2$), and T_1 and T_2 conflict, then $T_1 \longrightarrow_N T_2$.*

Proof (lemma 1): Since T_1 and T_2 conflict, they belong to the same conflict class. Since DISCOR enqueues transactions into the conflict class queues in the order they are delivered, and transactions are executed in the order in which they appear in the queues, DISCOR executes T_1 before T_2, and hence, $T_1 \longrightarrow_N T_2$. □

LEMMA 2 CONFLICT EQUIVALENCE (DISCOR). *For any two sites, N, N', running the* DISCOR *algorithm, history H_N produced by N is conflict equivalent to history $H_{N'}$ produced by N'.*

Proof: (lemma 2) Only transactions within the same conflict class conflict. From Lemma 1, if N is the master of a conflict class, all transactions in that class are ordered in H_N according to the delivery order at N. N FIFO-multicasts commit messages in the order these transactions are executed, and N' applies the updates of these transactions according to that order, and hence, they appear in $H_{N'}$ in the same order as in H_N. Similar reasoning applies for conflicting transactions whose master is N' according to the order N' executes and FIFO-multicasts them. For those remote conflicting transactions whose master is neither N nor N', both sites will apply their updates in the same order the master FIFO-multicasts them. 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 (DISCOR). *The history produced by the* DISCOR *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. □

A.2 Liveness of DISCOR

THEOREM 2 LIVENESS (DISCOR). *The* DISCOR *algorithm ensures that a transaction T_i delivered at an available site eventually commits.*

Proof: (theorem 2) Assume for contradiction that a transaction T_i is delivered at an available site but never committed in the absence of catastrophic failures. This can only be possible if either (a) the transaction is never executed by an available master site, or if (b) there is an infinite chain of master site failures before committing the transaction. Since delivery is reliable, if T_i is delivered in view V by a site available in V, then T_i is delivered by all available sites in V. Either the master of T_i in V is available, delivers T_i, append T_i to its conflict class, and eventually execute and commit it, or it fails before doing so. If it fails, a view change occurs, another site will become master for T_i. It is guaranteed that this site has delivered T_i, and hence, eventually commit T_i or fail. Unless all sites fail, eventually one master will be up long enough to execute and commit the transaction. □.

A.3 Consistency of DISCOR

Failed sites obviously do not deliver the same transactions as available sites. Let $\mathcal{T}$ be the subset of transactions committed at a site N before it failed.

THEOREM 3 CONSISTENCY OF FAILED SITES (DISCOR). *All transactions, T_i, $T_i \in \mathcal{T}$ are committed at all available sites. Moreover, the committed projection of the history in N is conflict equivalent to the committed projection of the history of any of the available sites when this history is restricted to the transactions in $\mathcal{T}$.*

Proof: (theorem 3) For a transaction to be committed anywhere, its commit message must have been delivered. Due to uniform reliable delivery, if any site delivers a commit message, all available sites will deliver the commit message. Thus, all commit messages of transactions in $\mathcal{T}$ have been delivered at all available sites. Since we are assuming there are no catastrophic failures, there is at least one available site that will also commit the transaction. The equivalence of histories follows directly from Lemma 1. □

A.4 Correctness of NODO

Again, we first only look at available sites. 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. In the following, we indicate with $T_1 \longrightarrow_{OPT} T_2$ that T_1 was OPT-delivered before T_2, and with $T_1 \longrightarrow_{TO} T_2$ that T_1 was TO-delivered before T_2.

DEFINITION 1 DIRECT CONFLICT. *Two transactions T_1 and T_2 are in direct conflict at site N if $T_1 \longrightarrow_N T_2$ and there is no transaction T_3 such that $T_1 \longrightarrow_N T_3 \longrightarrow_N T_2$.*

LEMMA 3 TOTAL ORDER AND SERIALIZABILITY (NODO). *Let H_N be the history at site N, let $T_1,\ T_2$ be two directly conflicting transactions in H_N. If $T_1 \longrightarrow_{TO} T_2$ then $T_1 \longrightarrow_N T_2$.*

Proof (lemma 3): Assume the lemma does not hold, i.e., there is a pair of transactions $T_1,\ T_2$ such that $T_1 \longrightarrow_N T_2$ 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 $T_1 \longrightarrow_N T_2$, then the entry for T_1 must have been ahead in the queue. Upon T_2's TO-delivery, however, NODO would have reordered T_2 before T_1 (and aborted T_1 if it was the first in the queue. This results in $T_2 \longrightarrow_N T_1$ which contradicts the initial assumption. □

LEMMA 4 CONFLICT EQUIVALENCE (NODO). *For any two sites, $N,\ N'$, running the* NODO *algorithm, H_N is conflict equivalent to $H_{N'}$.*

Proof: (lemma 4) From Lemma 3, 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 4 1CPSR (NODO). *The history produced by the* NODO *algorithm is one copy serializable.*

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

A.5 Liveness of NODO

THEOREM 5 LIVENESS (NODO). *The* NODO *algorithm ensures that each transaction T_i, TO-delivered at an available site, eventually commits in the absence of catastrophic failures.*

Proof: (theorem 5) 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 site 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 delivered the commit (note that the commit is not sent with uniform-reliable delivery), a new master will reexecute the transaction and resend the commit message. □

A.6 Consistency of NODO

Let $\mathcal{T}$ be the subset of transactions committed a site N before it failed.

THEOREM 6 CONSISTENCY OF FAILED SITES (NODO). *All transactions, T_i, $T_i \in \mathcal{T}$ are committed at all available sites. Moreover, the committed projection of the history in N is conflict equivalent to the committed projection of the history of any of the available sites when this history is restricted to the transactions in $\mathcal{T}$.*

Proof: (theorem 6) 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 (due to uniformity). N only commits a transaction after having received the commit message (independently of whether N is the master or not). If the available sites have received the commit message they also commit. If they have not received it because the master failed (commit is not sent with uniform reliable delivery), a new master takes over, executes the transactions again (in the same order as before), and sends the commit message. Again if the new master fails before sending the commit, another master will

take over and repeat the procedure. Since we assume that there are some available sites, eventually one of these sites will become the master and the transaction will commit. The equivalence of histories follows directly from Lemma 3. □

A.7 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. Again, we first assume no failures.

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 5 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 at each site N, $T_i \longrightarrow_N T_s$.*

Proof (lemma 5):
It follows trivially from the algorithm. □

LEMMA 6 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 6): 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 5 $T_i \longrightarrow T_s$. 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 $T_j \notin \mathcal{T}_{T_s}$, 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 7 CONFLICT EQUIVALENCE (REORDERING). *For any two sites, $N,\ N'$, running the* REORDERING *algorithm, H_N is conflict equivalent to $H_{N'}$.*

Proof: (lemma 7) 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 ($T_1 \longrightarrow_{OPT} T_2$), then, from Lemma 6, 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 (no reordering), then other sites cannot change this order. The argument is similar to that in Lemma 3 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 3, it follows that all sites must commit T_1 and T_2 in the same order.

• $C_{T_1} \subseteq C_{T_2}$ 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 sites, Lemma 6 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: $T_1 \ldots \longrightarrow_N T_i \longrightarrow_N T_{i+1} \longrightarrow_N \ldots T_2$, where each pair of transactions in that sequence is in 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 7 1CPSR (REORDERING). *The history produced by the* REORDERING *algorithm is one copy serializable.*

Proof: (theorem 7) From Lemma 7, all histories are conflict equivalent. Moreover, they are all serializable. Thus, the global history is one copy serializable. □

A.8 Liveness of REORDERING

THEOREM 8 LIVENESS (REORDERING). *The* REORDERING *algorithm ensures that each transaction TO-delivered at an available site T_i eventually commits in the absence of catastrophic failures.*

Proof: (theorem 8) 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 site, 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 T_j that is serialized by transaction T_k, $T_k \neq T_i$ 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.

If a transaction is TO-delivered at one site, it is TO-delivered at all available sites. Failures lead to masters reassignment but do not introduce different cases to the above ones. □

A.9 Consistency of REORDERING

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

THEOREM 9 CONSISTENCY OF FAILED SITES (REORDERING). *All transactions, T_i, $T_i \in \mathcal{T}$, that are committed at a failed site N are committed at all available sites. Moreover, the committed projection of the history in N, is conflict equivalent to the committed projection of the history of any of the available sites when this history is restricted to the transactions in $\mathcal{T}$.*

Proof: (theorem 9) Since both transaction and commit messages are sent with uniform reliable multicast, if any site commits a transaction (after receiving the commit message), all available sites will receive the commit message, and hence, commit the transaction. This guarantees that when any site commits a reordered transaction in a different order than the TO-delivery, all available sites will do so. If a master fails after having executed a reordered transaction but before sending the commit message, another site will take over as master. This site might not reorder the transaction or reorder it differently (because OPT-delivery was different at the new master). But this is fine because no site has yet committed the transaction (not even the old master).
To prove the equivalence of histories, the theorem follows directly from Lemma 7. □