Diss. ETH Nr. 13864

Database Replication for Clusters of Workstations

DISSERTATION

submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by

BETTINA KEMME

Diplom-Informatikerin Universität Erlangen-Nürnberg
citizen of Germany
born October 24, 1968

accepted on the recommendation of
Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. André Schiper, co-examiner

2000

Acknowledgments

First of all, I would like to thank my supervisor Prof. Gustavo Alonso. He has not only provided helpful guidance but also a lot of inspiration and motivation. I have learnt many things about research working together with him, and his support has been essential for the success of my work. I am also thankful to Prof. André Schiper, my co-examiner. The many discussions and the cooperation we had in the context of the DRAGON project have been very helpful for a better understanding of group communication systems and their relationship to database systems.

During the last years I have worked with many different people and this cooperation has had considerable influence on the research presented in this thesis. In particular, the many discussions with Fernando Pedone and Matthias Wiesmann from the EPF Lausanne helped to create my "big picture" of replication in database systems and group communication systems. Thanks also to Marta Patiño-Martínez and Ricardo Jiménez-Peris from the Technical University of Madrid for the cooperation and friendship during the last year. The work would have not been possible without many interested and engaged students who contributed in the form of semester and diploma works: Stefan Pleisch, Matthias Zwicker, Ignaz Bachmann, Win Bausch, Guido Riedweg, Christoph Oehler and Michael Baumer.

I would also like to thank all my colleagues from the Information and Communication Systems Group and from the Database Research Group for their friendliness, discussions, and help. Special thanks go to Guy Pardon for sharing an office with me for so many years – I think we have built quite a good team. I am also thankful to Antoinette Förster for keeping the business running.

Thanks also to all my friends for their motivating encouragement, for the stimulating discussions about research, career, and many other topics, and for being there when I needed them.

A thousand thanks to Mike for his love and support. With him at my side, work seemed less hard and private time a treasure. He supported my research tremendously, kept me motivated, and was patient when I was stressed and in a bad mood.

Finally, I dedicate this thesis to my family. They supported me in every respect, and were always willing to give me moral support.

Contents

Abstract

This thesis is centered around the topic *database replication*. The work has been motivated by advances in the development of cluster databases and their specific demands in terms of high throughput, low response times, flexible load-balancing, data consistency and fault-tolerance. *Eager, update everywhere replication* seems a promising mechanism to achieve these goals. By *replicating* data across the cluster, transactions have fast access to local copies. *Eager* replica control provides data consistency and fault-tolerance in a straightforward way. Furthermore, via an *update everywhere* approach, all types of transactions can be submitted at any site without restrictions. Despite these characteristics, eager update everywhere replication is rarely used in practice since existing solutions have severe disadvantages in terms of performance and complexity. The objective of this thesis has been to develop an eager, update everywhere replication tool that avoids these limitations of current solutions and achieves the performance needed in cluster databases.

At the beginning, we present a couple of basic techniques that we believe are necessary to provide efficiency: keeping the number of messages small, simplifying the coordination between the databases, avoiding distributed deadlocks and the need for a 2-phase commit protocol, and providing different levels of transaction isolation and fault-tolerance in order to adapt to various hardware and workload configurations. Some of these mechanisms are based on the powerful multicast primitives of group communication systems. In particular, we use the ordering and fault-tolerance services of these systems to support replica control and data consistency.

Based on these techniques, the thesis develops a replication tool in three steps. First, we develop a theoretical framework including a family of replica control protocols. These protocols provide different levels of isolation and fault-tolerance, and differ in the way they treat write operations. All protocols provide data consistency on available sites and correctness criteria that are well understood and widely used in practice.

As a second step, we evaluate our approach in two ways. First, we present a simulation based performance comparison of the proposed protocols and traditional solutions. This study shows that our approach provides superior performance in comparison to traditional protocols. By providing a family of protocols, it becomes more stable and can adjust to specific conditions like high network costs or high conflict rates. Second, we have proven the feasibility of the approach in a real cluster environment by integrating it into the database system PostgreSQL. Most of the functionality could be added in separate modules with only a few changes necessary to the existing PostgreSQL system. We are confident that the approach can be implemented in a similar way in other systems. The evaluation of the system verified the results of the simulation study and proved that eager update everywhere can achieve high throughputs and low response times, provide good scalability and allow for flexible load-balancing.

The third step of this thesis evaluates further important issues. We present a solution to recovery that smoothly fits into the proposed framework. In our approach, nodes can join the system without interrupting transaction processing in the rest of the system while preserving data consistency. Furthermore, we propose a partial replication scheme that attempts to keep the replication overhead proportional to the number of replica while still providing a simple and flexible replica control mechanism.

In summary, this thesis presents an eager update everywhere replication tool that provides most of the functionality needed in cluster databases. The approach has been developed by building a solid theoretical framework and by proving its feasibility in terms of performance (simulation-based) and practicability (with a real implementation).

Kurzfassung

Diese Dissertation befasst sich mit dem Thema *Datenbankreplikation* und wurde durch die bedeutsamen Fortschritte im Bereich der Cluster-Datenbanken motiviert. Cluster-Datenbanken zeichnen sich durch ihre hohen Anforderungen bezüglich Durchsatz, Antwortzeiten, flexibler Lastbalanzierung, Datenkonsistenz und Fehlertoleranz aus. *Synchrone, update-everywhere Replikation* scheint ein vielversprechender Mechanismus zu sein, um diese Ziele zu erreichen. Durch *Replikation* der Daten auf mehrere Rechner können Transaktionen schnell auf die lokalen Kopien zugreifen. Eine *synchrone* Replikationskontrolle ermöglicht nahezu automatisch Datenkonsistenz und Fehlertoleranz. Weiterhin können nur mit einem *update-everywhere* Ansatz Transaktionen an jedem beliebigen Rechner ausgeführt werden. Synchrone, update-everywhere Replikation wird jedoch in der Praxis so gut wie nicht eingesetzt. Grund dafür sind die schlechte Leistung und die Komplexität existierender Protokolle. Ziel dieser Dissertation ist es, ein Tool für synchrone, update-everywhere Replikation zu entwickeln, das die Nachteile bisheriger Ansätze vermeidet und die Leistung erreicht, die in Cluster-Datenbanken benötigt wird.

Zu Beginn der Arbeit werden eine Reihe von Techniken präsentiert, die sich als notwendig herauskristallisiert haben, um die geforderte Effizienz zu erreichen: so muss z.B. die Anzahl der Nachrichten so gering wie möglich sein, die Koordination zwischen den Datenbanksystemen muss nach einfachen Regeln ablaufen, und verteilte Deadlocks und der Einsatz von einem 2-Phasen-Commit Protokoll sollten vermieden werden. Einige der eingesetzten Mechanismen basieren auf leistungsstarken Kommunikationsprimitiven, wie sie von Gruppenkommunikationssystemen angeboten werden. Insbesondere werden die Ordnungs- und Fehlertoleranzsemantiken dieser Systeme verwendet, um Replikationskontrolle und Datenkonsistenz zu unterstützen.

Basierend auf diesen Techniken wird in dieser Arbeit ein Replikationssystem in drei Schritten aufgebaut. In einem ersten Schritt wird eine Familie von Replikationskontrollprotokollen entwickelt. Diese Protokolle bieten unterschiedliche Isolations- und Fehlertoleranzstufen an, mit denen sich das System an verschiedene Hardware-Konfigurationen und Arbeitslasten anpassen kann. Des weiteren ermöglichen sie unterschiedliche Behandlungsformen für Schreiboperationen. In jedem Falle garantieren die Protokolle Datenkonsistenz auf den verfügbaren Rechnern und bieten Korrektheitskriterien an, die nicht nur klar definiert sind, sondern auch in existierenden Systemen vielfach eingesetzt werden.

In einem zweiten Schritt wird der Ansatz auf zweierlei Weise evaluiert. Zum einen wird die Leistung der einzelnen Protokolle mittels einer Simulationsstudie verglichen, zum anderen ist eines der Protokolle in das existierende Datenbanksystem PostgreSQL integriert worden. Die Simulationsstudie zeigt, dass der in dieser Arbeit vorgeschlagene Ansatz deutlich höhere Leistungswerte als konventionelle Protokolle ermöglicht. Durch die Existenz einer ganzen Familie von Replikationsprotokollen ist er stabiler und kann auch bei hohen Netzwerkkosten und hohen Konfliktraten eingesetzt werden.

Die "reale" Implementierung, d.h. die Integrierung eines der Protokolle in ein existierendes Datenbanksystem hat gezeigt, dass der Ansatz auch praktikabel und umsetzbar ist. Der Grossteil der Funktionalität konnte in separaten Modulen zum Datenbanksystem hinzugefügt werden, und nur wenige Änderungen an Post-

greSQL direkt waren notwendig. Dies lässt die Vermutung zu, dass der vorgeschlagene Ansatz in ähnlicher Weise auch in andere Systeme integriert werden kann. Umfangreiche Leistungsmessungen haben die Resultate der Simulationsstudie verifiziert und gezeigt, dass synchrone, update-everywhere Replikation hohen Durchsatz und kurze Antwortzeiten anbieten kann, gute Skalierbarkeit aufweist, und flexible Lastbalanzierung ermöglicht.

In einem dritten Schritt diskutiert die Arbeit weitere wichtige Punkte. Es werden Lösungsvorschläge zum Thema Recovery erarbeitet, die sich nahtlos in den Gesamtansatz einbinden lassen. Insbesondere können Rechner ohne Unterbrechung des laufenden Betriebes in das System eingefügt werden. Zusätzlich wird das Thema der partiellen Replikation behandelt. Die Arbeit schlägt Mechanismen vor, die versuchen, den Aufwand der Replikationskontrolle pro Datenobjekt proportional zur Anzahl der Kopien zu halten.

Insgesamt präsentiert die Arbeit einen synchronen, update-everywhere Replikationsansatz, der einen grossen Teil der Funktionalität anbietet, die in Clusterkonfigurationen benötigt wird. Die Arbeit basiert auf einer soliden theoretischen Grundlage und validiert ihren Ansatz sowohl bezüglich Leistung (mittels Simulation) als auch Durchführbarkeit (mittels einer realen Implementierung).

1 Introduction

1.1 Motivation

Over the last years, the tremendous advances in communication technology have triggered far-reaching developments in distributed information systems. The rapid growth of the Internet, as well as the widespread use of cluster and mobile computing technology have opened up new opportunities for advanced database systems. For instance, challenging applications like electronic commerce, remote airline check-in, or telebanking have appeared only in the last years. In these environments, *data replication* is a key component for both fault-tolerance and efficiency. Replicating data across several sites improves fault-tolerance since available sites can take over the work of failed servers. Furthermore, reading local copies instead of remote data helps to reduce the response time and increase the throughput of the system. Replication, however, has also overhead: copies must be kept consistent and reflect the latest updates. Due to this overhead it is important to use adequate replication mechanisms.

Existing replication solutions can be divided into eager and lazy approaches [GHOS96]. *Eager* protocols propagate updates to remote copies within the transaction boundaries and coordinate the different sites before the transaction commits. With this, data consistency, transaction isolation and fault-tolerance are provided in a straightforward way. Furthermore, these protocols are very flexible since they usually allow any copy in the system to be updated. In spite of the large number of existing eager protocols [BHG87], few of these ideas have ever been used in commercial products. There is a strong belief among database designers that most existing solutions are not feasible due to their complexity, poor performance and lack of scalability [GHOS96]. In fact, current products adopt a very pragmatic approach: they mostly use *lazy* approaches, in which updates are propagated to remote copies only after transaction commit [Sta94, Gol94]. As a result, fault-tolerance is restricted, and copies can become stale and even inconsistent. Often, there is no good understanding of the degree of inconsistency that can occur and solving the inconsistencies is usually left to the user. In order to avoid these problems, there has been considerable research effort in the last years to develop lazy schemes that also provide consistency [CRR96, PMS99, BK97, ABKW98, BKR$^+$99]. The price to be paid is that updates must be performed at dedicated primary copies losing many of the advantages of replication (the primary becomes a bottleneck and a single point of failure). In addition, severe restrictions need to be introduced either on the sites a transaction can access and/or on the distribution of the copies across the system.

Compared with each other, existing eager and lazy solutions have somewhat complementary behavior. Eager protocols emphasize consistency and fault-tolerance, but show poor performance. Lazy replication pays more attention to efficiency at the price of restricting transaction execution or not providing adequate degrees of data consistency. Clearly, lazy replication is necessary when copies cannot be easily synchronized. Typical examples are mobile terminals but, in general, this is true of all environments where the communication costs are too high. We believe, however, that for a wide range of environments and systems, eager replication is not only the preferable option from a correctness point of view but is also a feasible alternative in terms of performance. For instance, eager replication is highly appropriate in *computer clusters* of the type used to build very large databases using many, off-the-shelf computers [RNS96]. Such architectures can be found behind many Internet sites and web-farms [Buy99]. In such an environment, eager replication can greatly facilitate the implementation of otherwise complex functionality such as automatic load balancing or high availability. However, to circumvent the limitations of traditional eager solutions, it is necessary to rethink the way transaction and replica management is done. Motivated by these observations, this thesis has aimed at bridging the gap between replication theory and practice by developing a tool that efficiently supports eager replication in cluster based databases.

1.2 About this Work

The work of this thesis has been done in the context of the DRAGON project, a joint effort between the Information and Communication Systems Research Group of ETH Zürich and the Laboratoire de Systèmes d'Exploitation (LSE) of the EPF Lausanne, Switzerland [1]. The broad objective of the project is to investigate the use of group communication primitives in database replication protocols. Within this project, the objective of this thesis has been to provide a practical solution to eager, update-every replication that guarantees consistency, provides clear correctness criteria and reasonable fault-tolerance, avoids bottlenecks, and has good performance.

Development of a Replication Framework In a first step we analyze existing eager and lazy solutions and their particular advantages and disadvantages. This analysis serves as a motivation to identify a series of techniques that we believe are necessary to circumvent the limitations of current solutions. The most important of them are an efficient and powerful communication system, and relaxed but well defined correctness criteria.

- Communication is a key issue in distributed computing, since efficiency can only be achieved when the communication overhead is small. Therefore, unlike previous replication protocols, our solution is tightly integrated with the underlying communication system. Following initial work in this area [AAES97, Alo97, PGS97], we exploit the semantics of *group communication* [HT93] in order to minimize the overhead. Group communication systems, such as Isis [BSS91], Transis [DM96], Totem [MMSA$^+$96], OGS [FGS98], or Horus [vRBM96], provide group maintenance, reliable message exchange, and message ordering primitives between a group of nodes. We use their ordering and fault-tolerance services to perform several tasks of the database, thereby avoiding some of the performance limitations of current replication protocols.
- A second important point is correctness. It is well known that serializability provides the highest correctness level but is too restrictive in practice. To address this problem, we support different *levels of isolation* as implemented by commercial systems [BBG$^+$95]. Similarly, the approach also provides different *levels of fault-tolerance* to allow different failure behaviors at different costs. Such correctness criteria provide flexibility since they guarantee correctness when needed and allow the relaxation of correctness when performance is the main issue. All of them are well defined, well understood, and widely accepted in practice. In all cases, however, our solutions guarantee data consistency.

The basic mechanism behind our protocols is to first perform a transaction locally, deferring and batching writes to remote copies until transaction commit time (or performing them on shadow copies). At commit

[1] DRAGON stands for Database Replication based on Group Communication. DRAGON is funded by the Swiss Federal Institute of Technology (ETHZ and EPFL).

time all updates (the write set) are sent to all copies using a total order multicast which guarantees that all nodes receive all write sets in exactly the same order. Upon reception, each site (including the local site) performs a conflict test checking for read/write conflicts. Only transactions passing this test can commit. Conflicting write operations are executed in arrival order, thereby serializing transactions. As a result, message overhead is small, the local execution of a transaction is decoupled from applying the changes at the remote sites, no explicit 2-phase-commit protocol is needed and deadlocks are avoided.

Based on this basic mechanism we develop a family of replication protocols that differ in the level of isolation, the level of fault-tolerance and whether write operations are deferred or executed on shadow copies. To analyze the protocols, we have developed a detailed simulation study. This allows us to compare the performance of all protocols under various conditions: fast and slow communication, varying number of sites, varying workload etc. The results show that our approach shows clearly better performance than traditional solutions and supports a wide spectrum of applications and configurations. A fault-tolerant, fully serializable protocol can be used under certain conditions (fast communication, low conflict rate). If the system configuration is not ideal, the optimizations in terms of lower levels of isolation and fault-tolerance help to maintain reasonable performance.

Implementation of a Replicated Database System While a simulation study is the adequate tool to provide a relative comparison of different protocols, only a "real" implementation allows us to evaluate the *feasibility* of our solution and test the system in a *real cluster environment*. To study these issues we have integrated our approach into the kernel of the object-relational database system PostgreSQL [Pos98]. We have named the enhanced system *Postgres-R*. We decided to enhance an existing system instead of building the replication tool on top of it because replication is only efficient when it is tightly coupled with the underlying database system.

This implementation provides helpful insights into the practical issues of turning an abstract solution into a working system. It has shown that we were able to adjust our approach to the specifics of PostgreSQL without any conceptual changes. One important aspect has been to find a modular approach that minimizes the changes that are necessary to the original system. In fact, a great part of our implementation could be added to PostgreSQL as separate modules and only few modules of PostgreSQL had to be changed. With this, we believe that our approach can be integrated into a variety of different systems.

We have performed extensive tests on Postgres-R and obtained detailed performance results. The results prove that eager replication is feasible in cluster configurations and can scale to a relatively large number of nodes. Postgres-R provides fast response times, supports high throughput rates, and can take advantage of the increasing processing capacity when new nodes are added to the cluster. This shows that the limitations of traditional eager solutions can be avoided. At the same time Postgres-R provides fault-tolerance, data consistency and the flexibility to execute a transaction at any site in the system – characteristics that cannot be provided by lazy replication.

Recovery A replicated system must not only be able to mask failures but also allow failed nodes to recover and rejoin the system. We have developed recovery strategies based on standard database techniques and group communication primitives that can easily be integrated into our replication framework. The suggested solution allows a node to (re-)join the system without barely any interruption of transaction processing at the other nodes. At the same time it recovers the database of the joining node in such a way that it does not miss the updates of any concurrent transaction. A simple version of the suggested solution is implemented in Postgres-R.

Partial Replication Not all applications need or can replicate all data at all sites. Instead, nodes can have non-replicated data, or data is only replicated at some of the nodes. Partial replication is needed to adjust unbalanced data access or to handle large databases. We explore partial replication in several dimensions. First, transactions that only access non-replicated data do not require any communication. Second, updates to data that is not replicated at all sites requires significant less processing power than fully replicated data. Furthermore, we provide a subscribe/unsubscribe scheme that facilitates to flexibly change the data distribution in the system. Last, we introduce distributed transactions that transparently access remote data if data is not stored locally. The feasibility of these concepts has been verified by integrating them into Postgres-R.

1.3 Structure of the Thesis

The structure of the thesis is as follows: Chapter 2 first presents an introduction to replica control. Then it analyzes the main problems and limitations of existing eager replication protocols and shows how we address these limitations. Chapter 3 describes the system model. A great part of this chapter is dedicated to group communication systems and the primitives they provide. The second part presents the transaction model. It also introduces the replica control model used by our approach. Chapter 4 presents a family of protocols. The first part develops protocols providing different levels of isolation and presents their proofs of correctness. The second part redefines the algorithms to provide different levels of fault-tolerance. The chapter also compares the presented protocols with previous work. Chapter 5 provides a simulation based evaluation of the presented protocols. Chapter 6 presents the architecture and implementation details of the replicated database system Postgres-R. Chapter 7 provides performance results of Postgres-R in a cluster environment. Chapter 8 discusses recovery and Chapter 9 is dedicated to partial replication. Finally, Chapter 10 concludes the thesis.

2 Overview

This chapter provides an informal introduction to database replication along with an overview of the advantages and disadvantages of traditional solutions. In particular, we elaborate on the main drawbacks of these solutions to be able to distinguish between inherent and avoidable limitations. This leads us to the key concepts behind the approach proposed in this dissertation which eliminate the avoidable and alleviate the inherent limitations of traditional solutions.

2.1 Basics of Replica Control

A replicated database system is a distributed system in which each site stores a copy of the database (full replication) or parts of the database (partial replication). Data access is done via transactions. A transaction represents a logical unit of read and write operations. Two important components of a replicated database system are concurrency control and replica control. *Concurrency control* isolates concurrent transactions with conflicting operations, while *replica control* coordinates the access to the different copies.

The strongest correctness criteria for a replicated system is *1-copy-serializability* (1CSR) [BHG87]: despite the existence of multiple copies, an object appears 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*). Furthermore, *transaction atomicity* guarantees that a transaction commits (executes successfully) on all or none of the participating sites despite the possibility of failures. Not all replica control protocols guarantee 1-copy-serializability or atomicity; some provide lower or undefined levels of correctness in order to increase performance.

Gray et al. [GHOS96] categorize replica control mechanisms according to two parameters: *when* updates are propagated between the copies, and *where* updates take place, i.e., which copies can be updated (Table 2.1).

Update propagation can be done within transaction boundaries or after transaction commit. In the first case, replication is *eager*, otherwise it is *lazy*. Eager replication allows the detection of conflicts before the transaction commits. This approach provides data consistency in a straightforward way, but the resulting communication overhead increases response times significantly. To keep response times short, lazy replication delays the propagation of changes until after the end of the transaction, implementing update propagation as a background process. However, since copies are allowed to diverge, inconsistencies might occur.

In terms of which copy to update, there are two possibilities: centralizing updates (*primary copy*) or a distributed approach (*update everywhere*). Using a primary copy approach, all updates on a certain data item are first performed at the primary copy of this data item and then propagated to the secondary copies. This avoids concurrent updates to different copies and simplifies concurrency control, but it also introduces a potential bottleneck and a single point of failure. Update everywhere allows any copy of a data item to be updated requiring the coordination of updates to the different copies. In eager schemes this leads to

<table>
<tr><th>when / where</th><th>Eager</th><th>Lazy</th></tr>
<tr><td rowspan="2">Primary Copy</td><td>Early Solutions in Ingres</td><td>Sybase/IBM/Oracle
Placement Strat.</td></tr>
<tr><td colspan="2">Serialization–Graph based</td></tr>
<tr><td>Update Everywhere</td><td>Quorum based
ROWA/ROWAA
Oracle Synchr. Repl.</td><td>Oracle/Sybase/IBM
Weak Consistency Strat.</td></tr>
</table>

Table 2.1: Classification of replica control mechanisms

expensive communication within the transaction execution time. In lazy schemes the non-trivial problem of reconciliation arises. When two transactions update different copies of the same data item and both commit locally before propagating the update, the data becomes inconsistent. Such a conflict must be detected and reconciled.

2.2 Traditional Eager Replication

Using eager replication, 1-copy-serializability and atomicity can be achieved in a straightforward way. Replica control is combined with the concurrency control mechanisms, for instance 2-phase-locking (2PL) or timestamp based algorithms, in order to guarantee serializability. Furthermore, an atomic commitment protocol, like 2-phase-commit (2PC) is run at the end of the transaction to provide atomicity.

Table 2.1 classifies some of the better known protocols. Early solutions, e.g., distributed INGRES, use synchronous primary copy/site approaches [AD76, Sto79]. Most of the algorithms avoid this centralized solution and follow the update everywhere approach guaranteeing 1-copy-equivalence by accessing a sufficient number of copies. A simple approach is *read-one/write-all (ROWA)* [BHG87], which requires update operations to access all copies while read operations are done locally. This approach has the major drawback of not being fault-tolerant: processing halts whenever a copy is not accessible. To tolerate site failures, *read-one/write-all-available* (ROWAA) is used, which requires to update only the available copies [BG84, GSC$^+$83]. Carey et. al [CL91] provide an evaluation of ROWA with different concurrency control mechanisms.

Alternatively, different types of *quorum* protocols require both read and write operations to access a quorum of copies or sites [Tho79, Gif79, JM87, PL88]. As long as a quorum of copies agrees on executing the operation, the operation can succeed. For instance, the quorum consensus algorithm [Gif79], requires two write quorums and a write and a read quorum to always have a common copy. In this way, at least one of the read copies will have the latest value. Other solutions combine ROWA/ROWAA with quorum protocols [ES83, ET89]. Good surveys of early solutions are [DGMS85, BHG87, CHKS94]. Similarly, a great deal of work has been devoted to minimize quorum sizes or communication costs, or analyze the trade-off between quorum sizes and fault-tolerance [Mae85, AE90, KS93, RST95, TP98].

In [AES97, HSAA00], the authors suggest to execute transactions locally in an optimistic way and send the updates using epidemic mechanisms that provide some form of semantic ordering between messages. If there are conflicting transactions concurrently active in the system (not ordered) specific mechanisms abort all but one of these transactions. The epidemic update propagation is similar to a 2-phase commit, however, its main purpose is to provide serializability and not atomicity. In [SAE98], multicast primitives with different ordering semantics are used to propagate updates. Also here, ordering messages helps to serialize transactions and to reduce abort rates but the algorithms still require an atomic commit protocol to guarantee serializability.

Despite the elegance of eager replication, only few commercial systems implement eager solutions. *Oracle Advanced Replication* [Ora97] provides an eager protocol which is implemented through stored procedures activated by triggers. An update is first executed locally and then "after row" triggers are used to synchronously propagate the changes and to lock the corresponding remote copies. Oracle's handbook, however, recommends to "use eager replication only when applications require that replicated sites remain continuously synchronized" and quickly points out the drawbacks: "a replication system that uses eager propagation of replication data is highly dependent on system and network availability because it can function only when all sites in the system are concurrently available". Most other eager solutions mainly focus on availability and represent highly specialized solutions (e.g., Tandem's Remote Duplicate Database Facility or Informix's High-Availability Data Replication). In all cases, a 2-phase commit protocol is executed at the end of the transaction.

2.3 Lazy Replication

Due to the complexity and performance implications of eager replication there exist a wide spectrum of lazy schemes. Naturally, lazy replication reduces response times since transactions can be executed and committed locally and only then updates are propagated to the other sites. However, 1-copy-serializability can only be guaranteed in very restricted primary copy configurations. Some lazy schemes only ensure that all replicas of a data item eventually converge to a single final value and do not consider that transactions create dependencies between the values of different data items. Atomicity cannot be guaranteed at all. If a node fails before it propagates the updates of a committed transaction T to the other sites, then T is lost. Many lazy schemes use a primary copy approach. This means that update transactions must be submitted at the site with the corresponding primary copies and transactions which want to update data items whose primary copies reside at different sites, are not allowed.

2.3.1 Replication in Commercial Databases

Clearly, commercial databases favor lazy propagation models (see Table 2.1). Most systems started with a primary copy approach specialized for either OLTP (On Line Transaction Processing) or OLAP (On Line Analytical Processing) [Sta94, Gol94]. In the meanwhile, many of the big database vendors provide a whole spectrum of primary copy and update everywhere approaches.

- *Sybase Replication Server* provides an extended publish-and-subscribe scheme and clearly favors a primary copy approach although update everywhere configurations are possible. Updates are propagated to the other copies immediately after the commit of the transaction. The updates are obtained from the log as soon as the log records are stored on disk. This *push* strategy is an effort to minimize the time that the copies are inconsistent and an implicit acknowledgment of the importance of keeping copies consistent in an OLTP environment. In the primary copy configuration, updates can either be done by synchronously connecting to the primary site or asynchronously by transferring procedure calls between the site that wants to update the item and the primary site. In their update everywhere configuration, updates may take place at any site and conflict resolution has to be done by the application.

- *IBM Data Propagator* was first a primary copy approach geared towards OLAP and mobile architectures. It adopted a *pull* strategy in which updates were propagated only at the client request, which implies that a client will not see its own updates unless it requests them from the central copy. Having OLAP applications in mind, those requests may range from simple point-in-time refreshes and continuous update notifications to sophisticated subscriptions for aggregate data. Over the years, IBM enhanced the system to also support update everywhere providing conflict detection and automatic compensation. IBM also uses the log information to detect updates, and, to optimize the process, can even capture log records directly from the memory of the database system.
- *Oracle Symmetric Replication* [Ora97] supports both push and pull strategies, as well as eager and lazy replication. It is based on triggers to detect replicated data and activate data transfer. Oracle's snapshot mechanism for OLAP workloads is a primary copy approach where snapshot copies are modified on a pull basis either doing a full refresh or an incremental update using special logs. To provide OLTP applications with fast update propagation, Oracle implements a push strategy through deferred triggers and persistent queues. Oracle also allows update everywhere replication. Conflict resolution can be done automatically using some of the mechanisms Oracle provides (e.g., highest timestamp) but it is left to the user to decide which one is the correct copy.

Common to most of these lazy approaches is the fact that copies are not kept consistent at all times, in spite of acknowledging the importance of consistency. This forces users to develop their applications taking into account the fact that the data may be obsolete.

2.3.2 Lazy Replication with Lower Levels of Consistency

From the research point of view, there has also been considerable work in lazy replication. Early papers provide the user with a way to control inconsistency, i.e., although the data may be obsolete or even inconsistent, the degree to which the data may be "wrong" is limited and well-defined. A couple of weak consistency models have been constructed that provide correctness criteria weaker than 1-copy-serializability. Examples of weak-consistency replication models are Epsilon-serializability [PL91] and N-Ignorance [KB91]. Epsilon-serializability measures the distance between database objects like the difference in value or the number of updates applied. The application can therefore specify the amount of inconsistency tolerated by a transaction. N-Ignorance is based on quorums. It relaxes the requirement that quorums must intersect in such a way that the inconsistencies introduced by concurrent transactions are bounded. The replication system in Mariposa [SAS$^+$96] builds an economic framework for data replication. The frequency of update propagation depends on how much the maintainer of a replica is willing to pay. Also the staleness of the data in a query is determined by the price a user wants to pay. For all these approaches, however, making the choice of the right bound of inconsistency is a non-trivial problem and users must have a good understanding of the inconsistency metrics.

2.3.3 Lazy Replication providing 1-Copy-Serializability

More recent work has explored the possibility of using lazy replication while still providing 1-copy-serializability. Thus, Chundi et al. [CRR96] have shown that even in lazy primary copy schemes, serializability cannot be guaranteed in every case. The way to get around this problem is to restrict the placement of primary and secondary copies across the system. The main idea is to define the set of allowed configurations using configuration graphs where nodes are the sites and there is a non-directed edge between two sites if one has the primary copy and the other a secondary copy for a given data item. If this graph is acyclic serializability can be guaranteed by simply propagating updates sometime after transaction commit [CRR96]. Pacitti et al. [PSM98, PMS99] have enhanced these initial results by allowing certain cyclic configurations. These configurations, however, require more complex update propagation schemes, namely, updates to secondary copies must be executed in causal or the same total order at all sites. Breitbart et al. [BKR+99] propose an alternative solution by requiring the directed configuration graph (edges are directed from primary copy to secondary copy) to have no cycles. This also requires to introduce more sophisticated update propagation strategies. One strategy transforms the graph into a tree where a primary copy is not necessarily directly connected with all its secondary copies but there exists a path from the primary to each secondary. Update propagation is then performed along the paths of the graph. A second strategy assigns timestamps to transactions, thereby defining a total order that can be used to update secondary copies. Breitbart et al. also propose an alternative scheme where lazy propagation is applied along the acyclic paths of the graph while eager replication is used whenever there are cycles.

Although the proposed solutions allow a rather wide spectrum of configurations, in real applications (and specially in clusters) the complexities and limitations on replica placement are likely to be a significant liability. Furthermore, the primary copy approach limits the types of transactions that are allowed to execute.

2.3.4 Combining Eager and Lazy Replication

A further primary copy approach combining eager and lazy techniques has been proposed in [BK97, ABKW98]. The system is eager since the serialization order is determined before the commit of the transactions (using distributed locking or a global serialization graph). This means that communication takes place within the execution time of each transaction. However, the system can be called lazy because within the boundaries of the transaction the execution of the operations only takes place at one site. Propagating the updates to the remote copies is only after the commit and there is no 2-phase commit.

2.3.5 Replication in Non-Database Systems

There exist many lazy replication solutions that have not evolved with the concept of transactions in mind but in a more general distributed setting, for instance, distributed file systems, replication on the web [RGK96], document replication [ARM97], and so forth. A good survey of early approaches can be found in [CP92]. In these environments, lazy replication provides more easily the requested level of consistency because transactional dependencies do not need to be considered.

2.4 Replication and Cluster Computing

In cluster computing, the workload is distributed among several off-the-shelf workstations connected by a fast network. Cluster computing is used for *scalability* and *fault-tolerance*. Whenever the workload increases more nodes are added to the system in order to increase the process capacity. Furthermore, cluster computing provides fault-tolerance if the failure of one site does not hinder the execution at the other sites. It might even be possible for the available nodes to take over the work of failed nodes.

If clusters are used for databases, a first option is to partition the data among the sites and each site executes the requests on this specific partition. Partitioning, however, has some severe restrictions that motivate the need for replication. First, it is difficult to divide the data in such a way that the workload can be equally distributed among the sites. Second, the problem of distributed transaction management arises if transactions want to access data of different partitions. Often, databases have a hot-spot area that is accessed by most of the transactions. This data cannot simply be assigned to a single site. Finally, the failures of individual nodes lead to the inaccessibility of the corresponding partitions.

Replicating the data can help to eliminate these restrictions if the proper replication strategy is chosen:

- *Load balancing* is possible and can adjust to changing user input as long as for most of the transactions there are no restrictions where to execute them. Since primary copy approaches restrict the execution of updating transactions to the primary node they might not be appropriate depending on the workload. In contrast, update everywhere allows transactions to be started at any site.
- *Distributed transaction management* can be avoided if all data is replicated at all sites. It some cases it might already help to replicate the hot-spot data at all sites. If data is not fully replicated, individual sites must be able to subscribe online in a fast and simple manner to individual data items in order to adjust to changing user access patterns.
- *Fault-tolerance* can be provided by using eager replication, since data is available as long as one copy is accessible.
- *Consistency* is extremely important. Since cluster computing aims in high throughput there is no time to handle inconsistencies manually. Instead, replica control must either be able to detect and correct inconsistencies automatically before the database copies diverge, or avoid them from the beginning.

As a result of these considerations, it is clear that in terms of functionality an eager, update everywhere approach is the most desirable form of replication for cluster computing: it allows for optimal load distribution and local data access, and guarantees consistency and fault-tolerance.

2.5 Problems of Traditional Eager Replication and how to avoid them

Although eager update everywhere replication is the adequate choice from a theoretical point of view, current solutions are not attractive options in terms of performance and complexity. The question to ask is whether their limitations are completely inherent to the eager, update everywhere model or whether some of them are only an artifact of the mechanisms used. In what follows, we discuss some of the typical mechanisms found in traditional approaches, how they influence performance and complexity, and how their drawbacks can be circumvented by applying adequate techniques.

2.5.1 Quorums

As noted before, many different quorum solutions have been proposed in the literature. The different approaches have been compared by mainly analyzing the communication costs, i.e., how many messages and how many message rounds are required, and the availability, i.e., how many failures can be tolerated without stopping the entire system. In general, quorums are considered an elegant mechanism to decrease message and execution overhead and to increase fault-tolerance. However, while they might be suitable for replicated file management and other forms of object management, we believe that the only feasible configuration in a real database system is a ROWA, respectively ROWAA, approach for two reasons: scalability and complexity of read operations.

Scalability Although quorums claim to decrease execution and communication overhead, a simple analysis shows that they do not scale, i.e., they do not behave well if there are many nodes in the system. In fact, any form of replication will only scale when the update ratio is small. And in this case, ROWA, respectively ROWAA, is the best approach.

Assume a centralized system having a *total capacity* of op_{total} operations per second. The entire capacity can be used to execute *local* transactions, therefore $op_{local} = op_{total}$. Assume now a system where the *number of nodes* is n, all of them identical to the centralized one (i.e., each one with a capacity of op_{total}). Assume that the *fraction of replicated data* is given by s, with data replicated at all n nodes. The *fraction of write accesses* is given by w (therefore *1-w* is the fraction of read operations). The number of copies (quorum) that must be accessed for a read operation is RT (*read threshold*) and for a write operation is WT (*write threshold*). We assume that the operations of a local transaction of a node N are executed at N itself and additionally at as many other nodes as necessary for the read/write threshold to be reached. Assuming that each node has the same probability to participate in a quorum, the probability that a remote operation is executed at a node N' is $\frac{RT-1}{n-1}$ and $\frac{WT-1}{n-1}$ for read and write operations respectively. This means the total capacity of a node is divided between local and remote operations:

$$\begin{aligned} op_{total} &= op_{local} + (n-1)(1-w)s\tfrac{RT-1}{n-1}op_{local} + (n-1)ws\tfrac{WT-1}{n-1}op_{local} \\ &= op_{local}(1 + s(RT-1) + ws(WT-RT)) \end{aligned}$$

Regarding remote read operations, e.g., there are $n-1$ nodes having $(1-w)*s$ read operations on replicated data and a fraction $\frac{RT-1}{n-1}$ of such remote read operation is executed at each node. Of course, the less remote operations a node has to perform the more of its capacity can be used to process local operations.

From this, the scaleup is calculated as the ratio of local operations of an n-node-system compared to a 1-node-system (i.e., the optimal scaleup is n).

$$S(n) = \frac{op_{local}(n)}{op_{local}(1)} = \frac{n}{1 + s(RT-1) + ws(WT-RT)}$$

We look at three quorum solutions. The ROWA approach is modeled by $RT_{ROWA} = 1, WT_{ROWA} = n$ (ROWA minimizes the number of reads). Using a quorum consensus approach [Gif79] that minimizes the number of writes we set $RT_{QC} = \frac{n}{2}, WT_{QC} = \frac{n}{2}+1$. In the last case, we assume the nodes can be structured logically as a $n = m$ x p grid. A read quorum consists of one replica from each column while a write quorum consists of all the replicas in one column plus one replica from each of the remaining columns [CAA90]. Assuming a square grid with $m = p = \sqrt{n}$ we have $RT_G = \sqrt{n}$ while $WT_G = 2\sqrt{n}$. This results in a scaleup of:

- $S_{ROWA}(n) = \frac{n}{1+ws(n-1)}$ for the ROWA approach,
- $S_{QC}(n) = \frac{n}{1+s(\frac{n}{2}-1)+ws}$ for quorum consensus, and
- $S_G(n) = \frac{n}{1+s(\sqrt{n}-1)+ws\sqrt{n})}$ for the grid approach.

In the first case, the scaleup has a maximum of n when $ws = 0$ (no updates or no replicated data), in the second and third case the scaleup is at a maximum when $s = 0$ (no replicated data). For all three approaches, the scaleup has a minimum when $ws = 1$ (all data is replicated and all operations are updates). Figure 2.1 shows the scaleup for a 50-node-system at different update rates for $s = 1.0$ and $s = 0.2$. For all types of quorums, high update rates lead to very poor scaleup. Although a small write quorum improves the performance, the requirement that quorums must always intersect limits the reasonable use of the capacity. If ws approaches 1, the total capacity of the system tends to be 1 for the ROWA approach, 2 for equivalent

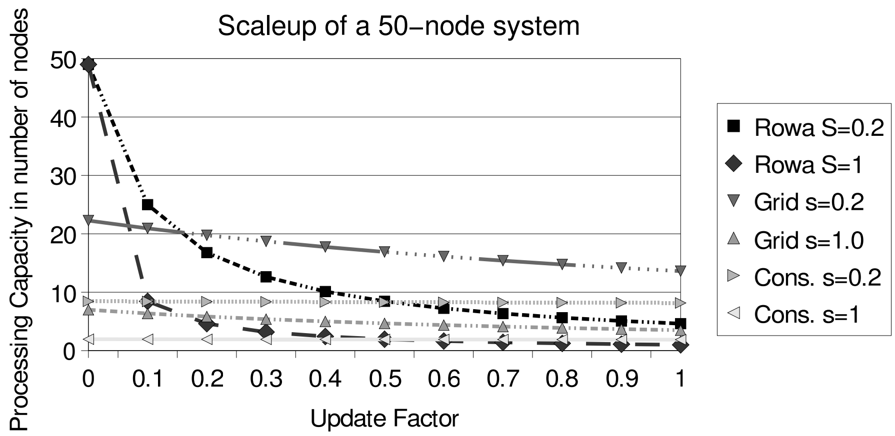

Figure 2.1: Scaleup for different update factors for a system with 50 nodes

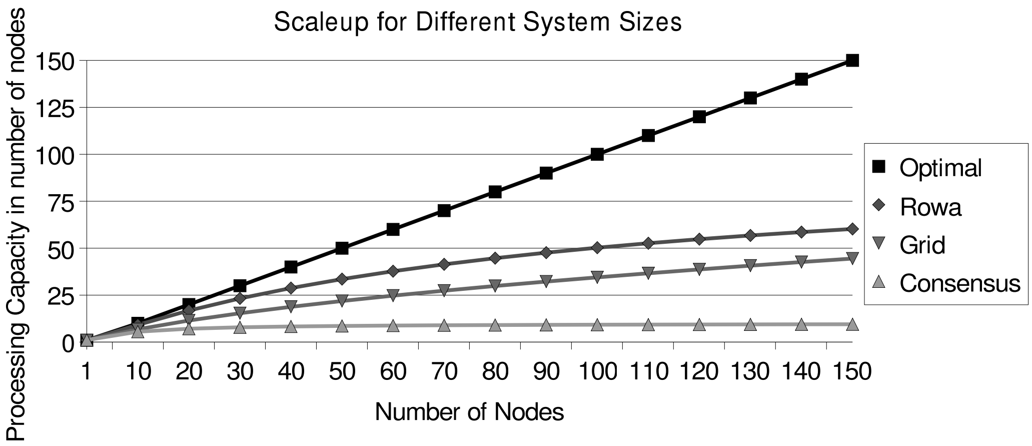

Figure 2.2: Scaleup for different numbers of nodes with ws factor of 0.02

sizes of read and write quorums, and 3.5 for the grid approach. Although the ROWA approach shows much better results for low update rates the drop in scaleup is still very sharp. For instance, if $s = 0.2$ and $w = 0.1$ total throughput is only half (25 nodes) the nominal capacity (50 nodes). Figure 2.2 depicts the scaleup for different system sizes at a replication factor of $s = 0.2$ and an update rate of $w = 0.05$ compared to the optimal scaleup. All three approaches scale very badly. In the tested configuration, ROWA performs better than the grid and the consensus approach since the update rate is small. However, one can observe that with increasing number of nodes the differences between ROWA and grid become smaller (beyond 400 nodes the grid protocol is better than ROWA). The reason is that although the update rate is constant, the absolute number of updates increases with the number of nodes favoring the grid protocol at large system sizes. Still, if we add more nodes and with it more transactions and more copies but keep a constant update and replication rate, the performance of the system degrades independently of the protocol.

The conclusions to draw from here are obvious: adding new nodes to the system in order to increase the transaction processing capacity is only then possible when the update rates are small. In such configurations, ROWA/ROWAA is the most appropriate approach from the point of view of scaleup.

Complexity of Read Operations The second reason why ROWA/ROWAA is the preferred approach in database systems becomes clear when we switch from the rather abstract model of read and write operations on single data items to real datasets and concrete interactions between user and database. Databases handle complex data items and not simple objects storing a single value. For instance, relational databases consist of tables each of them containing a set of records. The records themselves consist of several attributes. Update operations usually write one or more records (e.g., the SQL `update, delete` and `insert` statements) and the locking granularity for update operations is usually individual records. Reading operations, however, are often more complex (like, e.g., the SQL read operation `select`). They often scan all records of a table and determine a subset fulfilling a certain criteria to be returned to the user. It is not clear, how read quorums for such operations should be built and how should be determined which copy stores indeed the latest value. If the granularity for an update operation is a record and write quorums are smaller than the entire set of copies each site might have valid and stale records in a single table, and in fact, none of the sites might be able to return the correct result for a given query. ROWA/ROWAA approaches do not have this problem since all copies are up-to-date and read operations always only access the local data. Note that in other types of replicated systems, e.g., replicated file systems, the problem described here might not exist if the read request is a well defined access to a specific object (e.g. file) for which version maintenance is not complicated.

2.5.2 Message Overhead

In most traditional protocols, the separation between how to guarantee global serializability (e.g., by 2-phase-locking) and the problem of replica control implies that each access within a transaction is dealt with individually, i.e., an update operation incorporates a request and an acknowledgment per copy. This generates $2n$ messages, with n being the number of copies to be accessed, or 1 request and n acknowledgments if broadcast facilities are available. Clearly, this approach cannot scale beyond a few nodes. For example, in a 10-node-system, a throughput requirement of 100 transactions per second, where each transaction has around 5 update operations, results in 9,000 messages per second using point-to-point connections and 5.000 messages per second if broadcast is available. Such amount of network traffic is a potential major bottleneck. Furthermore, each of these messages represents a synchronization point among the sites. While the local site has to wait for the remote sites to return the acknowledgments, the transaction cannot proceed. Even in algorithms where the local site continues processing, each message produces context switches on all sites, thereby increasing complexity and reducing performance.

To minimize overhead and complexity, our approach follows a different strategy. We bundle write operations in a single *write set message* instead of treating them individually, a strategy commonly used in lazy replication. We do this by postponing all writes to replicated data to the end of the transaction (or performing them on shadow copies) and only then propagate the updates to the remote sites. With this, we greatly reduce the message overhead, the conflict profile of the transactions and the number of synchronization points within a transaction.

Furthermore, as noted before, replication is only effective when there is a significant number of read operations in the system, requiring a ROWA/ROWAA solution. This means not only that read operations need only be performed locally, but also, that sites should not even know anything about the read operations of remote transactions, not perform any version tests, etc., and no information about read operations should be included in messages.

2.5.3 Conflicts and Deadlocks

A further problem with traditional eager, update everywhere protocols is the conflict rate and probability of *deadlocks*. This problem was analyzed in detail by Gray et al. [GHOS96], who provided several approximations to the probability of deadlocks in different scenarios, showing that in some configurations the probability of deadlock is directly proportional to n^3, n being the number of replicas. This can be intuitively explained based on the fact that, as more transactions try to access the same objects and there are more replicas, the longer it takes to lock them all, thereby increasing the duration of the transactions. Furthermore, transactions also need longer to execute due to the communication overhead mentioned above. This also leads to higher conflict and deadlock rates.

A possible way to reduce conflicts and to avoid deadlocks is to pre-order the transactions. One way of doing this is to use *group communication systems* [HT93] to multicast messages within a group of sites. With group communication, it is possible to ensure that messages that are multicast within the group will be received at all sites in the same total order. Note that a site sends a message also to itself in order to be able to determine this total order. If all operations of a transaction are sent in a single write set message and write sets arrive at all sites in the same order, it suffices to grant the locks in the order the write sets arrive to guarantee that all sites perform the same updates and in exactly the same order. Additionally, transactions never get into a deadlock. Note that the total order on the delivery of write sets does not imply a serial execution since non-conflicting operations can be executed in parallel. Only the access to the lock table is serial (as in conventional transaction managers).

2.5.4 Levels of Isolation

Although serializability is the traditional correctness criteria used in databases [EGLT76], it often restricts the execution of concurrent transactions too severely and, consequently, commercial databases use a whole suite of correctness criteria that allow *lower levels of transaction isolation*. Some of them are defined in the SQL standard [ANS92, GR93, BBG$^+$95]. The different levels are a trade-off between correctness and performance in an attempt to maximize the degree of concurrency by reducing the conflict profile of transactions. They significantly improve performance but allow some forms of inconsistencies. Hence, they are somehow comparable to lazy solutions. However, they have the advantage of being more understandable and well accepted among database practitioners. We will see that using these relaxed correctness requirements is even more important in a distributed environment where the prolonged transaction execution times lead to higher conflict rates.

2.5.5 Fault-Tolerance

Most traditional protocols introduce a considerable amount of additional overhead during normal processing in order to provide fault-tolerance. In particular, they use commit protocols to guarantee the atomicity of a transaction on all sites in the system. Besides of being complex and time-consuming, such a commit protocol is a severe synchronization point between the different sites and has the effect that the response time of a transaction is determined by the slowest machine. This is so because a site only agrees on committing a transaction when it is able to guarantee the commitment locally, and this is typically considered the point when the entire transaction is successfully executed. Due to this time consuming synchronization, eager replication is mostly used in very specialized fault-tolerance configurations with dedicated hardware equipment.

As it is done with different levels of isolation, we propose to weaken full correctness to provide faster solutions. We allow a local site to commit a transaction whenever the global serialization order has been determined and do not require that it waits for the other sites to execute the transaction. Instead, the local site relies on the fact that the other sites will serialize the transaction in the same way according to the total order in which write sets are delivered.

Furthermore, we exploit the different degrees of reliable message delivery provided by group communication systems in order to determine the overall correctness in failure cases. While complex message exchange mechanisms guarantee the consistency on all sites beyond failures, faster communication protocols provide a best effort approach in which consistency is only guaranteed on non-faulty sites. This trade-off is very similar to that of 1-safe and 2-safe configurations typically found in back-up systems [GR93, AAE^{+}96].

3 Model

This chapter discusses the formal model and the tools on which the replication approach suggested in this dissertation are based. The description is divided in two parts. The communication model covers all those components and concepts of group communication primitives that are useful for the purpose of database replication. The transaction model described the main database concepts relevant in the context of this thesis. Furthermore, it introduces the basics of our replica control approach.

3.1 General Architecture

A distributed database consists of a number of *nodes*, $N_i, (0 < i \leq n)$, also called *sites* or *servers*. Nodes communicate with each other by exchanging *messages*. The system is asynchronous, i.e., different sites may run at different rates, and the delay of messages is unknown, possibly varying from one message to the next. In regard to failures, we assume a crash failure model [NT88]. A node runs correctly until it fails. From then on, it does not take any additional processing steps until it recovers. Communication links may also fail and stop transferring messages. We first assume a fully replicated system. In Chapter 9, we relax the requirement of full replication. The database consists of a set of objects $\mathcal{X}$, and each node N_i maintains a copy X_i of each $X \in \mathcal{X}$. We refer to X as the logical object and to X_i residing on node N_i as a physical copy. Users interact with the database by connecting to any of the nodes and submitting *transactions* to this node.

Figure 3.1 depicts the main components of a node [BHG87, GR93]. The *communication system* is responsible for reliably transferring messages among the sites. The *transaction manager* is responsible for executing transactions and uses some form of *concurrency control* to guarantee the isolation of concurrent transactions. In our approach, these three components are also responsible for *replica control*. They will be explained in detail in this chapter. The *data and recovery manager* is responsible for the data access and for the recovery of the database after a failure or at start time. It is discussed in Chapter 8.

3.2 Communication Model

The communication model is based on multicast primitives as implemented in group communication systems [BSS91, CT91, MMSA$^+$96, DM96, vRBM96, FGS98]. Group communication systems manage the message exchange (multicast) between groups of nodes and offer a variety of primitives. These primitives provide different *message ordering* semantics, *group maintenance*, and different degrees of *reliability*. We will use the different types of message ordering to serialize transactions while the group maintenance and reliability degrees will help to handle failures and recovery.

Figure 3.2 depicts the layered configuration we use. On each site, the application (in our case the database system), sends and receives messages by passing them to the communication module. From there, they are

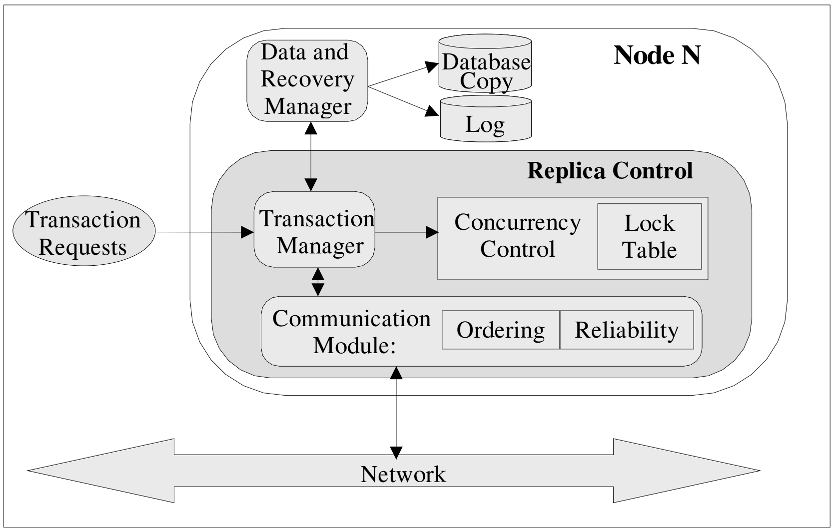

Figure 3.1: Architecture of a node in the replicated database system

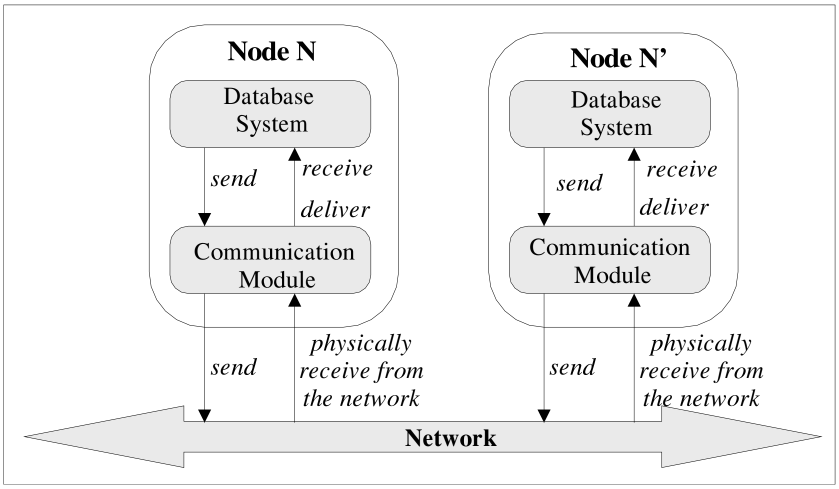

Figure 3.2: Communication module

forwarded to all sites including the sending site. This can be done via consecutive point-to-point messages or with a physical broadcast. We say that a node *multicasts* or *sends* a message to all group members. The communication module on each node N of the group then physically *receives* the message from the network and *delivers* the message to the local application. At this time the local application at N *receives* the message. In the following, we use the terms "a message is delivered at node N" and "node N receives a message" interchangeably with reference to the reception of the message at the application. The time at which the message is delivered to the application depends on the semantics of the primitive chosen. Note that, generally, the delivery of the same message at the different nodes does not take place at the same time: at one site a message might already be delivered while on another node the communication module has not even yet received the message from the network. In addition, group communication systems usually handle message loss transparently by implementing certain retransmission protocols. Hence, we assume that a message will eventually be delivered at all sites as long as sender, receivers and communication links are available for sufficient long time.

3.2.1 Message Ordering

An important feature is *message ordering*. The concept of ordering is motivated by the fact that different messages might depend on each other. For instance, if a node multicasts two messages, the content of the second message might depend on the content of the first, and hence, all nodes should receive the messages in the order they were sent (FIFO). Even stronger, a message m sent by node N might also depend on a message m' that N previously received. Formally, *causal precedence* is defined as the reflexive and transitive closure of: m causally precedes m' if a node N receives m before it sends m' or if a node N sends m before sending m' [BJ87b, SES89]. This means, a *causal order* multicast guarantees that the delivery order at each site respects causal precedence. FIFO and causal orders multicast primitives do not impose any delivery order on messages that are not causally related. Thus, different nodes might receive unrelated messages in different orders. A total order prevents this by ensuring that at all nodes all messages are delivered in the same serial order. In some implementations, the total order includes the FIFO or causal order.

To summarize, group communication systems provide multicast services which differ in the final order in which messages are delivered at each node N [HT93]:

I. *Basic* service: A message is delivered whenever it is physically received from the network. Thus, each node receives the messages in arbitrary order.
II. *FIFO* service: If a node sends message *m* before message *m'*, then no node receives *m'* unless it has previously received *m*.
III. *Causal* order service: If a message *m* causally precedes a message *m'* then no node receives *m'* until it has previously received *m*.
IV. *Total* order service: All messages are delivered in the same total order at all sites, i.e., if any two nodes *N* and *N'* receive some messages *m* and *m'*, then either both receive *m* before *m'* or both receive *m'* before *m*.

3.2.2 Fault-Tolerance

Node failures (and communication failures) lead to the exclusion of the unreachable nodes and are mainly detected using timeout protocols [CT91]. We will assume that in the case of network partitions only a primary partition may continue to work while the other partitions stop working and hence, behave like failed nodes. Within this section we only discuss failures. The joining of new nodes or nodes that previously failed will be discussed in detail in Chapter 8. The group communication system provides the application with a *virtual synchronous* view of these failure events in the system [BJ87a].

To achieve virtual synchrony, the communication module of each site maintains a view V_i of the current members of the group and each message is tagged with the identifier of the view in which it is sent. Whenever the group communication system observes a change in the number of nodes, it runs a coordination protocol called *view change protocol*, to agree on the delivery of messages that were sent in the current view V_i. This protocol guarantees that the communication system will deliver exactly the same messages at all non-failed members of view V_i. Only then is the new view V_{i+1} installed excluding failed nodes. The application is informed via a so called *view change message*. Hence, the application instances on the different sites perceive changes at the same virtual time. Since we consider a primary view system, we say

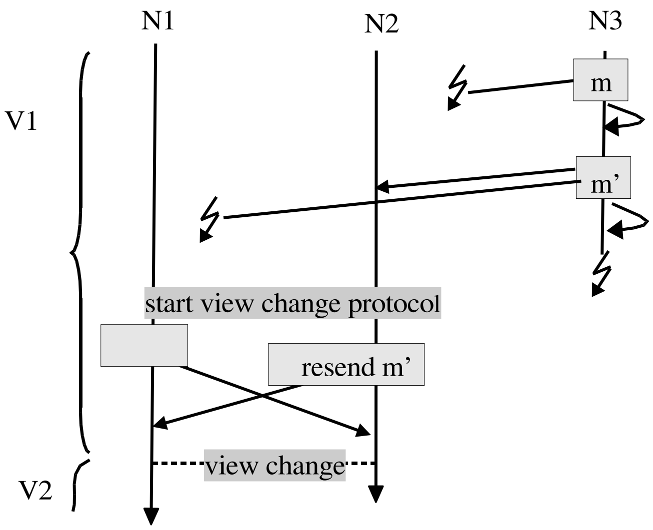

Figure 3.3: Example of a view change

that all sites of primary view V_i change to the same consecutive primary view V_{i+1} unless they fail or are disconnected from the majority of nodes in V_i. Figure 3.3 shows an example with three nodes. The vertical lines show the runs at the communication modules at nodes N_1, N_2 and N_3, where N_3 sends two messages m and m' in view V_i. Assume these messages are sent with the basic service (no ordering requirement). When N_3 fails, N_1 and N_2 run the view change protocol which detects that N_2 but not N_1 has physically received m' from the network. Hence, N_2 forwards m' to N_1 and both deliver m' before they install the new view.

Using virtual synchronous communication, the application at each node N sees a sequence of communication events. An event is either *sending a message* m, *receiving a message* m (this is also denoted as message m *is delivered* at N) or receiving a *view change* notification V_i. Only when the node is in a primary view V_i it can send or receive messages. We assume the first communication event on each site N to be a view change including N in the primary view. A run R_i of the application at node N, $N \in V_i$ and V_i primary view, is the sequence of communication events at N starting with primary view message V_i and ending either with installing consecutive primary view message V_{i+1} (we then say N is V_i-available) or the crash of N or N's exclusion due to a network partition (we then say N fails during V_i or that N is faulty in V_i). Based on the definitions in [BJ87a, HT93] and particularly in [SS93], virtual synchrony and the concept of reliability provides the following guarantees:

I. *View Synchrony*: if a message *m* sent by node *N* in view V_i is delivered at node *N'*, then *N'* receives *m* during R_i.
II. *Liveness:* if a message *m* is sent by node *N* in view V_i and *N* is V_i-available, then *m* is delivered at all V_i-available nodes.
III. *Reliability:* Two degrees of reliability are provided.
    1. *Reliable delivery:* if a message *m* sent in view V_i is delivered at V_i-available node *N'*, then *m* is delivered at all V_i-available nodes.
    2. *Uniform reliable delivery:* if a message *m* sent in view V_i is delivered at any node *N'* (i.e., *N'* is either V_i-available or fails during V_i), then *m* is delivered at all V_i-available nodes.

Uniform reliability and (simple) reliability differ in the messages that might be delivered at failed nodes.

With uniform reliable delivery, whenever a message m is delivered at any node N in view V_i (N might fail during V_i or be available until the next view is installed) m must be delivered at all V_i-available nodes (guarantee III.2). This means a node N receives the same messages as all other nodes *until* it fails [SS93] and the set of messages delivered at a failed node N is a subset of the messages delivered at the surviving nodes. With reliable delivery failed nodes might have received (and processed) messages that no other node receives. This means a node N receives the same messages as all other nodes *unless* it fails. V_i-available sites, however, receive exactly the same messages during V_i for both reliable and uniform reliable delivery.

As an example of the difference between reliable and uniform reliable delivery consider message m sent by node N_3 in Figure 3.3. In the case of reliable delivery, the communication module of N_3 will deliver m immediately back to its own application although none of the other sites has physically received the message. When N_3 now fails, not being able to resend m, the other sites will not deliver it. Using uniform reliable delivery, N_3 will not immediately deliver the message but would only do it when it knew that everybody else has received it.

In regard to ordering properties in the case of failures, existing group communication systems behave differently. In all systems, V_i-available sites will always deliver all messages in the correct order (FIFO, causal, or total). Furthermore, failures will not leave gaps in a causally related sequence of multicasts. That is, if m causally precedes m', m' is only delivered if m is delivered. For sites failing during V_i, there exist different implementations. As far as we know, all systems providing uniform reliable delivery, correctly order all messages on all sites (including failed sites). This means in the case of the total order that the order of messages observed at the failed site is a prefix of the order seen by the surviving nodes. In the case of reliable delivery the guarantees provided by the different systems vary. As far as we know, FIFO and causal order primitives guarantee correct delivery order even at failed nodes. This is not the case with all total order protocols. Some of them allow that failed nodes have delivered messages in different order than they are ordered at the available nodes. This means that the order services described in the previous section only hold for available nodes.

3.2.3 Protocols and Overhead

There exists a variety of approaches to implement the different ordering primitives, the reliability degrees and the view change protocol. We will briefly describe some of them to provide the reader with an intuition of how the communication system works.

View Change In order to handle message loss and failures, the communication module on each site acknowledges the reception of a multicast message from the network. Furthermore, the communication module keeps a message until it is *stable*, i.e., until it has received the acknowledgments for the message from all sites. With this, in a simple view change protocol [BSS91], each available node sends all *unstable* messages to all other available nodes (called *flush*) and waits until it receives their flush messages. Hence, all available sites have exactly the same messages and will deliver the same set of messages before delivering the view change message. In [SS93], the authors present coordinator based view change protocols for both reliable and uniform reliable message delivery that are also based on relaying unstable messages. A more rigorous approach to implement view change, based on a transformation to a consensus problem, is given in [GS96].

Whether all received messages are delivered or whether some messages are skipped depends strongly on the dependencies between messages. As an example, consider N_3 of Figure 3.3 which first sends messages m and m', and then fails. The remaining system receives m' but m was lost on the network. Assume now, N_3

has used the FIFO or causal order service. In these cases, during the view change protocol, N_1 and N_2 will not deliver m' but skip it because causal precedence forbids to deliver m' if m is not delivered. Of course, there must be a mechanism to detect that a message is missing. A message that cannot be delivered due to gaps in preceding messages is sometimes referred to as "orphan" [BSS91]. Note, that orphans are always messages sent by failed nodes. In contrast, if N_3 has used the basic service or a total order service that does not include a FIFO or causal order, then N_1 and N_2 would deliver m'.

Regarding failures, crashed nodes and disconnected nodes must behave the same. A crashed node simply stops processing. When nodes become disconnected from the current primary view V_i they usually do not stop but will also run a view change protocol detecting that they are in a minority view v_p. In the case of uniform reliable delivery care has to be taken that no message is delivered that will not be delivered in the primary view V_i. As an example, assume a system with 5 nodes in the primary view V_i that partition into N_1, N_2 and N_3 (that will build the consecutive primary view V_{i+1}), and N_4 and N_5 (that will install a minority view v_p). Now assume that a message m was sent before the partition was detected and only N_1 received the message. Similarly, a message m' was sent and only N_5 received it. While running the view change protocol, N_1, N_2 and N_3 will exchange message m, and N_4 and N_5 will exchange m'. Since N_1, N_2 and N_3 will move to the consecutive primary view they are allowed to deliver message m during the view change protocol, although N_4 and N_5 will not deliver it. This is correct because N_4 and N_5 are considered faulty and the delivery guarantee only refers to V_i-available nodes. On the other hand, N_4 and N_5 are not allowed to deliver m' because m' is not delivered at V_i-available nodes N_1, N_2 and N_3.

There exist a couple of special cases which must be considered, e.g., if node failures occur during the view change protocol or if different nodes perceive different failures or joins and start the view change protocol concurrently. Messages that are sent by the application after the view change protocol started, are usually delayed until the new view is installed. Numerous view change protocols have been proposed which always work with specific implementations of the causal and total order services and provide different degrees of reliability [BSS91, SS93, MAMSA94, FvR95b, GS96]. Since the view change might introduce a significant overhead, these systems only work when failures occur at a lower rate than messages exchange. Inter-failure intervals in the range of minutes or hours are therefore acceptable.

Basic/FIFO Multicast For the basic/FIFO order multicast, a message is simply broadcast to all sites [CT96, SR96]. Each site tags its messages with sequence numbers. Message loss is usually detected by a combination of timeouts for acknowledgments and gaps in the sequence numbers of received messages. If a site detects a gap in the sequence numbers of received multicast messages it sends a negative acknowledgment to the sender which then resends the message as a point-to-point message.

If reliable multicast is used, each site can deliver the message to the application immediately after its reception from the network (basic service) or after all preceding messages from the same sender (FIFO) have been delivered. Using uniform reliable multicast the communication module cannot deliver a message until it is stable, i.e., until it has received the acknowledgments from a sufficient number of sites. With this, the delivery of the message is delayed for at least an additional message round. This is similar to a 2-phase-commit, although it does not involve a voting process. As example, in Figure 3.4, a reliable multicast would deliver m at N_1 at timepoint 1, at N_2 at timepoint 2 and at N_3 at timepoint 3. Uniform reliable delivery would deliver m at N_1 only during the view change (since N_3 failed it cannot send the acknowledgment anymore), at N_2 at timepoint 4 (having now the acknowledgment from N_3) and at N_3 at timepoint 5.

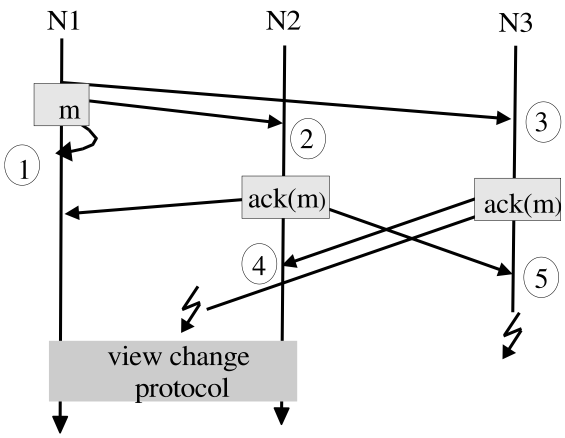

Figure 3.4: Examples for reliable and uniform reliable delivery

Causal Order There exist many solutions for implementing causal order. The basic idea is, that each message carries information about all causally preceding messages so that each site can independently determine whether it has already received and delivered all preceding messages. For instance, if we restrict the communication to broadcast, a message m contains in its header an array of the size of the number of nodes in the system. This array contains for each node N the identifier of the last message received from N before m was sent. Example algorithms can be found in [SES89, BSS91, HT93].

Total Order Multicast There are many alternatives to implement the total order primitive. Totem [MMSA$^+$96], e.g., uses a token: only when a node has the *token*, it is allowed to multicast messages. In the case the token holder crashes, Totem automatically generates a new token along with the view change protocol. Several application messages can be bundled into a single multicast message. The token carries a counter which is incremented for each message sent and the messages are labeled with the value of the counter. Therefore, messages have unique sequence numbers and their order can be determined directly on reception (*born-ordered messages*). Message loss can easily be detected by gaps in the sequence numbers of received messages. Acknowledgments are also piggybacked on the token. Before a site forwards the token it updates a field in the token which indicates the sequence number s of a message m confirming that it has received m and all messages with smaller sequence numbers than s. In the case of message loss, negative acknowledgments are sent, piggybacked on the total order multicast messages if possible. Another technique providing born-ordered messages is *sequencing*. One node in the system is a sequencer. All other nodes send their messages to this node, where they are stamped with sequence numbers and broadcast to all other nodes. This approach has originally been taken by the Amoeba distributed and replicated file system [KT96]. The ISIS/Horus/Ensemble system also uses a centralized approach where a *master* node decides on the order of the messages [BSS91, vRBM96, Hay98]. While each node broadcasts independently its messages to group members, received messages may only be delivered when an ordering message is received from the master. In this case, messages are not born-ordered. The problem of the last two approaches is that in the case of the failure of the master/sequencer, nodes might deliver messages in different order if reliable delivery is used. For instance, if the master sends an ordering message informing that it orders m before m', delivers m and m' to its own application but fails before the other sites receive the ordering message, the available sites might decide to deliver the messages in reverse order. Except for the token approach, acknowledgments are usually sent individually or piggybacked on other multicast messages.

Again, in the case of reliable multicast a message is delivered immediately after all preceding messages have been delivered. Uniform reliable message delivery additionally waits until sufficient nodes have acknowledged the reception of the message. In the case of the master approach the sites have to wait until sufficient nodes have acknowledged the reception of the ordering message.

In contrast to these three approaches where a single node decides on the message order without interaction with the rest of the system, a couple of distributed ordering mechanisms have been proposed where all nodes are included in the ordering process. These are, e.g., the atomic broadcast mechanisms based on the consensus protocol [CT96]. In its basic version the atomic broadcast is a coordinator based voting process. A message m is first sent to all sites using reliable multicast. After the physical reception of m, a node sends a proposal on the ordering of m to the coordinator. After having received enough proposals the coordinator decides on an order. If reliable delivery is used it sends a decision message with reliable multicast and the sites deliver m once they have received the decision message. In the case of uniform delivery the coordinator sends a decision to all sites which in return acknowledge the reception of the decision to the coordinator. Once the coordinator has received sufficient acknowledgments it reliably multicasts a delivery command. Upon reception of this delivery command a site delivers m. Besides this basic algorithm there exist many algorithms providing various optimizations in regard to the number of message rounds and the number of messages needed [Sch97, PS98, PS99] .

Cost Analysis Table 3.1 summarizes the costs of the presented algorithms in terms of the number of physical messages per multicast message and number of message rounds before a multicast message can be delivered. n indicates the number of nodes in the system. Note that the reliable multicast primitives are presented without the acknowledgments although in practice, they are also needed. We have omitted them to indicate that they need not be sent before the message is delivered to the application but anytime afterwards. Except for the token protocol, we assume that acknowledgments are sent in individual messages. The costs depend on the ordering semantics, on the existence of a broadcast medium, on the delivery degree, and – of course – on the chosen algorithm.

Clearly, the basic, FIFO and causal services are cheaper than the total order service which requires further communication in order to determine the total order.

Comparing for each algorithm the number of messages with and without a broadcast facility, we can see the importance of a broadcast medium. Especially the uniform reliable versions of the protocols require a tremendous amount of messages in point-to-point networks. Interestingly, using Atomic Broadcast, the differences are not as big and we never have a quadratic number of messages. The reason is the coordinator based acknowledgment of messages. Acknowledgments are only sent to the master which forwards one final message once it has received enough acknowledgments. All the other approaches (except the token) send decentralized acknowledgments (every nodes sends a message to every node). While coordinator based algorithms lead to less messages (in the case of point-to-point networks) they require more message rounds.

Comparing reliable delivery with uniform reliable delivery we clearly can see the high amount of messages needed for uniformity. Piggybacking acknowledgments would lead to less messages but higher delays, since messages are not delivered before the acknowledgments are received (see, for instance, the high number of message rounds for the token based protocol).

Comparing the different algorithms, we believe that each of them has its advantages in certain configurations. Although the token based approach requires many message rounds the token provides a natural

Protocol	Number messages broadcast medium	Number messages point-to-point	Message Rounds
Basic/FIFO/causal order			
reliable	1	n-1	1
unif. reliable	n	n(n-1)	2
Token based			
reliable	2	n+1	n/2
unif. reliable	2	n+1	n+n/2
Sequencer			
reliable	2	n+1	2
unif. reliable	n+1	n^2	3
Master			
reliable	2	2n	2
unif. reliable	n+1	n(n-1)	3
Atomic Broadcast			
reliable	n+1	3(n-1)	3
unif. reliable	2n+1	5(n-1)	5

Table 3.1: Cost analysis for different multicast algorithms

network access control method which might be useful at high message rates. Although the master approach requires more messages than the sequencer for point-to-point messages, one has to take into consideration that the sequencer approach sends a message twice (once to the sequencer and then from the sequencer to all sites) while in the master approach the message is only sent once (directly to all sites). Thus, for large messages, the master approach might be preferable. Although the atomic broadcast has more message rounds than the master and sequencer approach, it has advantages in point-to-point networks. Furthermore, we have only presented the basic mechanism, and many optimizations exist.

3.3 Transaction Model

Our transaction model follows the traditional one [BHG87]. A transaction T_i is a partial order, $<_i$, of read $r_i(X)$ and write $w_i(X)$ operations on logical objects X. When a transaction reads or writes a logical object more than once during its execution, the operations are indexed accordingly (e.g., if a transaction reads an object twice, the two operations are labeled $r_{i1}(X)$ and $r_{i2}(X)$ respectively.). We delimit a transaction T_i by introducing specific *begin of transaction* $BOT(T_i)$ and *end of transaction* $EOT(T_i)$ operations. Our replicated database model uses a *read-one/write-all-available* approach in which each logical read operation $r_i(X)$ is translated to a physical read $r_i(X_j)$ on the copy of the local node N_j. A write operation $w_i(X)$ is translated to physical writes $w_i(X_1), \ldots, w_i(X_n)$ on all (available) copies.

Database systems provide a couple of guarantees regarding transactions of which the following are of interest in the context of this thesis:

- *Isolation* guarantees that concurrent transactions are isolated from each other whenever necessary to guarantee consistency of transactions and database.
- *Atomicity* guarantees that a transaction T_i either commits, ($EOT(T_i) = c_i$), or aborts, ($EOT(T_i) =$

a_i), its results at all participating sites. This should be true in the failure-free case and when node or communication failures occur.

- *Durability* guarantees that the results of committed transactions are persistent. In the context of database replication this requires that a recovering or new site has to apply the updates of all transactions that were executed during its down-time.

3.3.1 Isolation

Transactions must be isolated from each other when they have conflicting operations. Operations conflict if they are from different transactions, access the same copy and at least one of them is a write. A history H is a partial order, $<_H$, of the physical operations of a set of transactions $\mathcal{T}$ such that $\forall T_i \in \mathcal{T}$: *if* $o_i(X) <_i o_i(Y), o \in \{r, w\}$, *then* $\forall o_i(X_j), o_i(Y_{j'}) \in H : o_i(X_j) <_H o_i(Y_{j'})$. Furthermore, all conflicting operations contained in H must be ordered.

To isolate conflicting transactions different *levels of isolation* are used [GLPT76, GR93]. The highest isolation level, *conflict serializability*, requires a history to be conflict-equivalent to a serial history, i.e., to a history in which transactions are executed serially. Lower levels of isolation are less restrictive but allow inconsistencies. The inconsistencies that might occur are defined in terms of several phenomena. The ANSI SQL standard specifies four degrees of isolation [ANS92]. However, recent work has shown that many protocols implemented in commercial systems and proposed in the literature do not match the ANSI isolation levels [BBG+95, Ady99, ALO00]. In the following, we adopt the notation of [BBG+95] and describe the phenomena of interest for this thesis as:

P1 *Dirty read:* $w_1(X_i) \ldots r_2(X_i) \ldots (c_1$ or $a_1)$. T_2 reads an uncommitted version of X. The most severe problem of dirty reads is cascading aborts. If T_1 aborts $w_1(X_i)$ will be undone. Hence, T_2 must also be aborted since it has read an invalid version.

P2 *Lost Update:* $r_1(X_i) \ldots w_2(X_i) \ldots w_1(X_i) \ldots c_1$. T_1 writes X upon the result of its read operation but not considering the new version of X created by T_2. T_2's update is lost.

P3 *Non-Repeatable Read:* $r_{11}(X_i) \ldots w_2(X_i) \ldots c_2 \ldots r_{12}(X_i)$. T_1 reads two different values of X.

P4 *Read Skew:* $r_1(X_i) \ldots w_2(X_i) \ldots w_2(Y_j) \ldots c_2 \ldots r_1(Y_j)$. If there exists a constraint between X and Y (e.g, the sum of both must be positive), T_1 might read versions of X and Y that do not fulfill this constraint.

P5 *Write Skew:* $r_1(X_i) \ldots r_1(Y_j) \ldots r_2(X_i) \ldots r_2(Y_j) \ldots w_1(Y_j) \ldots w_2(X_i)$. If there exists a constraint between X and Y (e.g. the sum of both must be positive) it might be violated by the two writes.

3.3.2 Concurrency Control Mechanisms

In most systems, locking protocols [BHG87, GR93] are used to implement the different isolation levels. Before reading an object, a shared read lock is acquired; before writing an object, an exclusive write lock is acquired. Since write locks are usually not released until commit time for recovery purposes (to avoid P1), the only possibility to reduce the conflict profile of a transaction is to release read locks as early as possible or to not get read locks at all. Accordingly, the protocols proposed in this thesis are based on different concurrency control mechanisms providing different levels of isolation. In all cases an exclusive write lock on object X is acquired before executing the corresponding operation $w(X)$ and only released at EOT.

Serializability implemented via strict 2-phase-locking [BHG87] avoids all phenomena described above. Both read and write locks are acquired before the execution of the corresponding operation and held until the end of transaction (called long locks).

Cursor Stability uses short read locks. Still, a read lock on object X is acquired before executing the operation $r(X)$. However, the lock is released directly after the operation is executed (short read lock). Only when the transaction wants to update the same object later on, the lock is kept and, upon executing the write operation, upgraded into an exclusive write lock.

With this, cursor stability avoids long delays of writers due to conflicts with read operations [GR93, BBG$^+$95, ALO00] and is widely implemented in commercial systems. For instance, if transactions scan the database and perform complex read operations, long read locks would block a big part of the database and delay write operations considerably. Short read locks avoid this behavior. However, inconsistencies might occur. To avoid the most severe of these inconsistencies, lost update ($P2$), the system keeps long read locks on objects that the transaction wants to update later on. For instance, SQL *cursors* keep a lock on the object that is currently referred to by the cursor. The lock is usually held until the cursor moves on or it is closed. If the object is updated, however, the lock is transformed into a write lock that is kept until EOT [BBG$^+$95]. As another example, the reading SQL `select`-statement can be called with a ``` ``for update'' ``` clause, and read locks will not be released after the operation but kept until EOT. Hence, if the transaction later submits an update operation on the same object, no other transaction can write the object in between. We call these mechanisms cursor stability in analogy to the discussion in [BBG$^+$95]. Note that write locks are still long, avoiding $P1$. Lost updates ($P2$) can be avoided by using cursors or the `for update` clause. Cases $P3$ to $P5$ might occur.

Snapshot Isolation is a way to eliminate read locks completely [BBG$^+$95]. Transactions read consistent snapshots from the log instead of from the current database. The updates of a transaction are integrated into the snapshot. Snapshot isolation uses object versions to provide individual snapshots. Each object version is labeled with the transaction that created the version. A transaction T reads the version of an object X labeled with the latest transaction which updated X and committed before T started. This version is *reconstructed* applying *undo* to the actual version of X until the requested version is generated. Before a transaction writes an object X, it performs a *version check*. If X was updated after T started, T will be aborted. This feature is called *first committer wins*. Snapshot isolation avoids $P1 - P4$ but allows $P5$. $P1$ is avoided because of the long write locks, $P2$ is avoided because the second writer is aborted, and $P3$ and $P4$ are avoided because read operations only read values committed before the transaction started. Snapshot isolation has been provided by Oracle since version 7.3 [Ora95] and has recently received more attention [SW99, SW00, ALO00]. Note that snapshot isolation avoids all inconsistencies described in the ANSI SQL standard although formally it does not provide serializable histories.

Hybrid Protocol combines 2-phase-locking and snapshot isolation by acquiring long read and write locks for update transactions. Read-only transactions use a snapshot of the database [SA93]. This protocol, unlike cursor stability and snapshot isolation, provides full serializability but requires transactions to be declared update or read-only in advance.

3.3.3 Failure Handling in Database Systems

Databases deal with failures based on the notion of consistency and atomicity and must be able to perform recovery.

1-copy-equivalence means that the multiple copies of an object appear as one logical object. *1-copy-equivalence* must also be provided in the case of failure. *Atomicity* implies that a transaction must either commit or abort at all nodes. Therefore, after the detection of a failure, the remaining nodes have to agree on what to do with pending transactions. This usually requires to execute a *termination protocol* among the remaining nodes before processing can be resumed [BHG87]. The termination protocol has to guarantee that, for each transaction the failed node might have committed/aborted, the remaining available nodes decide on the same outcome. Care has also to be taken of transactions that were still active on a site when it fails. If the rest of the system commits them it must be guaranteed that this decision was correct (the failed node was willing to commit them, too). This approach is very similar to the view change protocol in distributed systems. Whereas replicated database systems decide on the outcome of pending transactions, group communication systems decide on the delivery of pending messages. Hence, comparing typical protocols in both areas [BHG87, FvR95b], we believe the overhead to be similar. A significant difference is that database systems decide negatively – abort everything that does not need to be committed – while group communication systems behave positively – deliver as many messages as possible.

Failed nodes must also be able to *recover* and rejoin the system. When a failed node N restarts it first has to bring its own database copy into a consistent state. All updates of aborted transactions that are still reflected in the database must be undone. Furthermore, all updates of committed transactions that are not reflected in the database must be redone. In a second step, N has to update its database copy to reflect the updates of the transactions that have been processed in the system during the down time of N. Only then can it resume executing transactions. Similarly, a new node must receive a current database copy before it can start executing transactions. These data transfers to joining nodes must be coordinated with concurrent transaction processing in such a way that the joining nodes do not miss the updates of any transaction.

3.3.4 Execution and Replica Control Model

As described in Section 2.5, we will use a variation of the *read-one/write-all-available* (ROWAA) approach. All the updates of a transaction will be grouped in a single message, and the total order of the communication system will be used to order the transactions. A transaction T_i can be submitted to any node N in the system. T_i is *local* at N and N is called the *owner* or *local node* of T_i. For all other nodes, T_i is a *remote* transaction.

The execution of a transaction T_i is depicted in Figure 3.5. It takes place in five phases. During the *read phase*, T_i performs all read operations on the replicas of its owner N. The write operations are either deferred until all read operations have been executed or are performed on shadow copies. The second alternative enables to check consistency constraints, to capture write-read dependencies (i.e., a transaction can read what it has written previously) and to fire triggers. In any case, the write operations are bundled into a single *write set message* WS_i. When the transaction wants to commit, the *send phase* starts. The write set WS_i is sent to all available nodes (including the local node) using the total order multicast. This total order is used to determine the serialization order whenever transactions conflict. Upon the reception of WS_i at a node (local or remote) the *lock phase* of T_i is initiated. The transaction manager performs a test checking for read/write conflicts and takes appropriate actions. Furthermore, it orders conflicting write operations according to the total order determined by the communication system. This is done by requesting write locks for all write operations in a write set WS_i in an atomic step before the next write set is processed. The

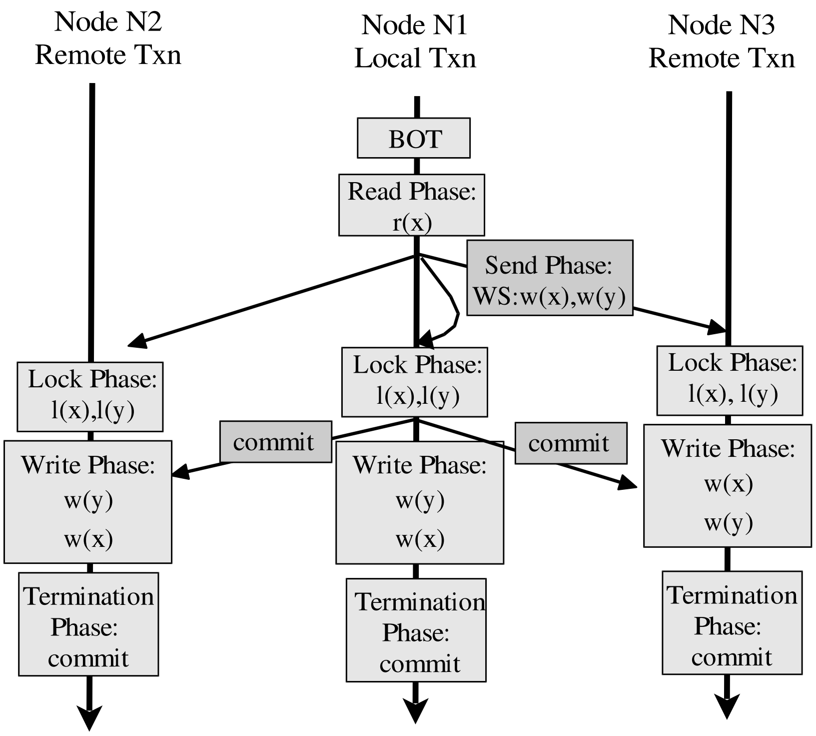

Figure 3.5: The 5-phase execution model of a transaction

processing of lock requests in the order of message delivery guarantees that conflicting write operations are ordered in the same way at all sites. Once all lock requests have been enqueued in the lock table (waiting or granted), the *write phase* starts. A transaction will only start the execution of an operation on a data item after all previous conflicting operations have been executed. Note that only the lock phases (requesting the write locks) of the different transactions are serial, but not the execution of the transactions. As long as operations of successive transactions do not conflict, their write phases may interleave. In the case a transaction has executed the write operations on shadow copies during the read phase, the local node does not need to reexecute the write operation but can install the shadow copies as the valid versions of the data items. In the *termination phase*, the transaction either commits or aborts. As mentioned in Section 2.5.5, although all sites terminate the transaction in the same way they will not run a 2-phase-commit protocol.

All the protocols proposed in the next chapter follow the scheme described and serialize write/write conflicts according to the total order. The differences between the protocols arise from how they check for read/write conflicts, how they handle them and how the termination phases look like.

4 A Suite of Replication Protocols

In this chapter we propose a suite of replica control protocols. For each protocol, we present the algorithm, discuss advantages and disadvantages, describe implementation issues, and provide correctness proofs. The first section presents protocols that differ in their levels of isolation using different concurrency control mechanisms. The second section proves the correctness of these protocols in the failure free case. The last section covers failures prone environments

4.1 Protocols with different Levels of Isolation

4.1.1 Serializability (SER-D)

We construct a 1-copy-serializable protocol by using a replicated version of strict 2-phase-locking (SER). This protocol is a modified version of one of the protocols described in [AAES97]. In this first version of the protocol, write operations are not executed immediately when they are submitted but deferred until the end of the transaction. Thus, in the shortcut we append a D, i.e. SER-D. Only read operations are executed immediately. Read/write conflicts are detected and handled during the lock phase. Whenever a write set is received, a conflict test checks for read/write conflicts between local transactions and the received write set. If the write set intersects with the read set of a concurrent local transaction which is still in its read or send phase the read transaction is aborted.

The protocol is shown in Figure 4.1 and it can be best explained through the example of Figure 4.2. The vertical lines in Figure 4.2 show a run at the two nodes N_1 and N_2. Both nodes have copies of objects X and Y. T_i reads X and then writes Y. T_j reads Y and then writes X. Both transactions are submitted at around the same time and start the local reads setting the corresponding read locks (read phase I in Figure 4.1). After the read phase, both transactions multicast independently their write sets WS_i and WS_j (send phase II). The communication system orders WS_i before WS_j. We first look at node N_1. When WS_i is delivered, the lock manager requests all locks for WS_i in an atomic step. Since no conflicting locks are active, the write lock on Y_1 is granted (III.1.a). From herein, the system will not abort the transaction. This is denoted with the c attached to the lock entries of T_i. The commit message c_i is multicast without any ordering requirement (III.2) and then the operation is executed (write phase IV). T_i can commit on N_1 once c_i is delivered and all operations have been performed (termination phase V.1). When WS_j is delivered, the lock manager processes the lock phase of WS_j. The lock for X_1 must wait (III.1.b). However, it is assured that it only must wait a finite time since T_i has already successfully received all necessary locks and can therefore commit and release the locks (note that the lock phase ends when the lock is included in the queue). We now have a look at node N_2. The lock manager also first requests all locks for WS_i (lock phase III). When requesting the lock for $w_i(Y_2)$ the lock manager encounters a read lock of T_j (conflict test). Since T_j is still in the send phase (WS_j has not yet been delivered) T_j is aborted and the lock is granted to T_i (III.1.c). If T_j were not aborted but instead $w_i(Y_2)$ waited for T_j to finish, we would observe a write skew phenomenon (on

The lock manager of each node N controls and coordinates the operation requests of a transaction T_i in the following manner:

I. *Local Read Phase:*
   1. *Read operations:* Acquire local read lock for each read operation $r_i(X)$ and execute the operation.
   2. *Write operations:* Defer write requests $w_i(X)$ until end of the read phase.

II. *Send Phase:* If T_i is read-only, then commit. Else bundle all writes in write set WS_i and multicast it (total order service).

III. *Lock Phase:* Upon delivery of WS_i, request all locks for WS_i in an atomic step:
   1. For each operation $w_i(X)$ in WS_i:
      a. If there is no lock on X or T_i is the only one owning locks, grant the lock.
      b. If there is a write lock on X or all read locks on X are from transactions that have already processed their lock phase, then enqueue the lock request.
      c. If there is a granted read lock $r_j(X)$ and the write set WS_j of T_j has not yet been delivered, abort T_j and grant $w_i(X)$. If WS_j has already been sent, then multicast abort message a_j (basic service).
   2. If T_i is a local transaction, multicast commit message c_i (basic service).

IV. *Write Phase:* Whenever a write lock is granted perform the corresponding operation.

V. *Termination Phase:*
   1. *Upon delivery of a commit message c_i:* Whenever all operations have been executed commit T_i and release all locks.
   2. *Upon delivery of an abort message a_i:* Undo all operations already executed and release all locks.

Figure 4.1: Replica control guaranteeing serializability using deferred writes (SER-D)

N_1 there is $r_i(X_1) <_H w_j(X_1)$, on N_2 there is $r_j(Y_2) <_H w_i(Y_2)$). This would result in a non-serializable execution that is not locally seen by any node. To avoid this problem, N_2 aborts its local transaction T_j. Since WS_j was already sent, N_2 sends an abort message a_j (no ordering required). When the lock manager of N_2 receives WS_j, it simply ignores it. Once a lock manager receives the decision message for T_i/T_j it terminates the transaction accordingly (V.1, V.2).

In this protocol, the execution of a transaction T_i requires two messages. One to multicast the write set (using the total order) and another with the decision to abort or commit (using the basic service). The second message is necessary since only the owner of T_i knows about the read operations of T_i and, therefore, about a possible abort of T_i. Once the owner of T_i has processed the lock phase for WS_i, T_i will commit as soon as the commit message arrives. Due to the actions initiated during the lock phase, it is guaranteed that remote nodes can obey the commit/abort decision of the owner of a transaction. When granting waiting locks, write locks should be given preference over read locks to avoid unnecessary aborts.

Several mechanisms could be used to avoid aborting the reading transaction. One possibility is to abort the writer instead of the reader. However, this would require a 2-phase-commit protocol since a writer could be aborted anywhere in the system. Assume a read/write conflict occurs on node N_i between T_i and T_j, T_j being a remote transaction submitted at node N_j and N_i decides to abort T_j. Since such a conflict can occur at any site and is not visible in the rest of the system (r_i is local at N_i), coordination in form of a 2-phase-commit is necessary. This coordination can only take place after all sites have completed the locking phase. This is exactly what our approach tries to avoid by enabling each site to independently decide on the outcome of exactly its local transactions. A second option would be to use traditional 2-phase-locking, i.e., the write operation has to wait until the reading transaction terminates. This introduces the possibility of global

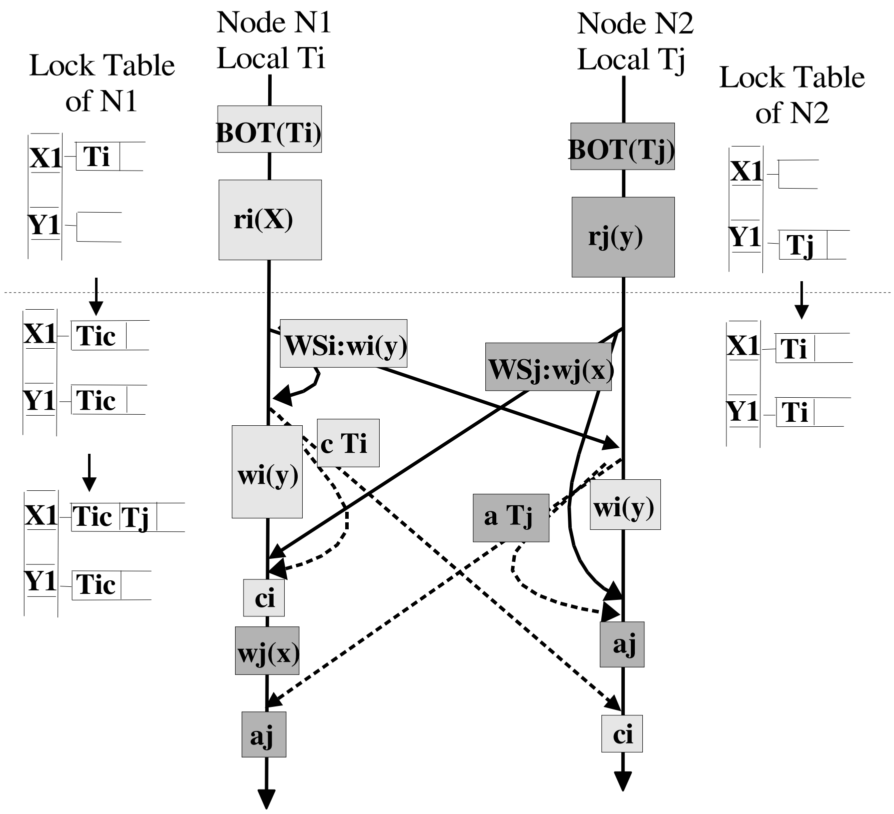

Figure 4.2: Example execution with the SER-D protocol

deadlocks, which are hard to detect, and again forbids the local site to decide independently on the outcome of a transaction. The third alternative is that each node informs the other nodes about local read operations so that each site can check individually whether read/write conflicts lead to non-serializable executions. The only way to do this efficiently would be to send information about the read set together with the write set. This information, however, is rather complex since it must not only contain the identifiers of all read objects but also which transactions had written the versions that were read (see, for instance, [AAES97]). As a result, messages are much larger and sites have significant more overhead to execute remote transactions.

As a summary, we would like to emphasize that although aborting readers is expensive, it is a straightforward way to provide 1-copy-serializability, to avoid distributed deadlocks, to keep reads local, and to avoid having to use 2-phase-commit. However, it might pay off not to abort read operations. The next protocols explore how this can be done.

4.1.2 Cursor Stability (CS-D)

The weak point of the SER protocol is that it aborts read operations when they conflict with writes. The protocols may even lead to starvation of reading transactions if they are continuously aborted. A simple and widely used solution to this problem is cursor stability (CS); this allows the early release of read locks. In this way, read operations will not be affected too much by writes, although the resulting execution may not

I.1	*Local Read Phase / Read operations:* Acquire local read lock for each read operation $r_i(X)$. In case of short locks (i.e., T_i will not update X later on), release the lock after the execution of the operation, otherwise keep it.
III.1.b	If there is write lock on X or all read locks on X are either short read locks or from transactions that have already processed their lock phase, then enqueue the lock request.

Figure 4.3: Protocol changes: cursor stability (CS-D)

be serializable.

The algorithm described in the previous section can be extended in a straightforward way to include short read locks. Figure 4.3 shows the steps that need to be changed in regard to the SER protocol in order to provide cursor stability based on deferred writes (CS-D). Step I.1 now requires short read locks to be released immediately after the object is read. Hence, the modified step III.1.b shows that CS does not need to abort upon read/write conflicts with short read locks since it is guaranteed that the write operation only waits a finite time for the lock. Upon read/write conflicts with long read locks aborts are still necessary. Note that when granting waiting locks, short read locks need not be delayed when write sets arrive but can be granted in the order they are requested since they do not lead to an abort. How far the protocol can really avoid aborts depends on the relation between short and long read locks; this is strongly application dependent. Like the SER protocol, the CS protocol requires two messages: the write set sent via a total order multicast and the decision message using a basic multicast.

The CS protocol fulfills the atomicity requirement of a transaction in the same way as the SER protocol does. It does not provide serializability but avoids $P1$ and $P2$. $P3$, $P4$ and $P5$ may occur.

4.1.3 Snapshot Isolation (SI-D)

Although CS solves the abort problem of SER, it generates inconsistencies. The problem is that read/write conflicts are difficult to handle since they are only visible at one site and not in the entire system. Snapshot isolation (SI) effectively separates read and write operations thereby avoiding read/write conflicts entirely. This has the advantage of allowing queries (read-only transactions) to be performed without interfering with updates.

In order to enforce the "first committer wins" rule, as well as to give appropriate snapshots to the readers, object versions must be labeled with transactions and transactions must be tagged with BOT (begin of transaction) and EOT (end of transaction) timestamps. The BOT timestamp determines which snapshot to access and does not need to be unique. The EOT timestamp indicates which transaction did what changes (created which object versions), and hence, must be unique. Oracle [BJK$^+$97] timestamps transactions using a counter of committed transactions. In a distributed environment, the difficulty is that the timestamps must be consistent at all sites. To achieve this, we use the sequence numbers of WS messages. Since write sets are delivered in the same order at all sites the sequence number of a write set is easy to determine, unique and identical across the system. Therefore, we set the EOT timestamp $TS_i(EOT)$ of transaction T_i to be the sequence number of its write set WS_i. The BOT timestamp $TS_i(BOT)$ is set to the highest sequence number of a message WS_j so that transaction T_j and all transactions whose WS have lower sequence numbers than WS_j have terminated. It is possible for transactions with higher sequence numbers to have committed but their changes will not be visible until all preceding transactions (with a lower message sequence number) have terminated.

Figure 4.4 describes the SI-D algorithm using deferred writes. We assume the amount of work to be done

The transaction manager of each node N coordinates the operation requests of the transactions as follows:

I. *Local Phase:*
   1. For each read operation $r_i(X)$, reconstruct the version of X labeled with T_j where T_j is the transaction with the highest $TS_j(EOT)$ so that $TS_j(EOT) \leq TS_i(BOT)$.
   2. Defer write operations.

II. *Send Phase:* If T_i is read-only, then commit. Else bundle all writes in WS_i and multicast it (total order service). The WS_i message also contains the $TS_i(BOT)$ timestamp.

III. *Lock and Version Check Phase:* Upon delivery of WS_i, perform in an atomic step:
For each operation $w_i(X)$ in WS_i:
   1. If there is no write lock on X and the current version of X is labeled with T_j. Then, if $TS_j(EOT) > TS_i(BOT)$, stop checking locks and abort T_i. Otherwise grant the lock.
   2. If there is a write lock on X or a write lock is waiting. Then let T_j be the last transaction to modify X before T_i: if $TS_j(EOT) > TS_i(BOT)$, then stop checking locks and abort T_i. Otherwise enqueue the lock request.

IV. *Write Phase:* Whenever a write lock is granted, perform the corresponding operation.

V. *Termination Phase:* Whenever all operations have been executed, commit T_i and release all locks.

Figure 4.4: Replica control based on snapshot isolation with deferred writes (SI-D)

for version reconstruction during the read phase I is small. Either this version is still available (Oracle, for instance, maintains several versions of an object [BJK$^+$97]) or it can be reconstructed by using a copy of the current version and applying undo until the requested version is generated. Oracle provides special rollback segments in main-memory to provide efficient undo [BJK$^+$97] and we will assume a similar mechanism is available. The lock phase III includes a version check. If $w_i(X) \in WS_i$, and X was updated by another transaction since T_i started, T_i will be aborted. We assume the version check to be a fast operation (the check occurs only for those data items that are updated by the transaction). In addition, to reduce the overhead in case of frequent aborts, a node can do a preliminary check on each local transaction T_i before it sends WS_i. If there already exists a write conflict with another transaction, T_i can be aborted and restarted locally. However, the check must be repeated upon reception of WS_i on each node.

With this algorithm, each node can decide locally, without having to communicate with other nodes, whether a transaction will be committed or aborted at all nodes. No extra decision message is necessary since conflicts only exist between write operations. The write set is the only message to be multicast.

While SER aborts readers when a conflicting write arrives, SI aborts all but one concurrent writers accessing the same item. We can therefore surmise that, regarding the abort rate, the advantages of one or the other algorithm will depend on the ratio between read and write operations. However, SI has some other advantages compared to SER or CS. It only requires a single multicast message to be sent and has the property that read operations do not interfere with write operations.

The SI protocol guarantees the atomicity of transactions. However, it does not provide serializability. It avoids $P1 - P4$ but $P5$ might occur.

4.1.4 Hybrid Protocol

SI provides queries with a consistent view of the database. However, update transactions might not be serializable. Moreover, if objects are updated frequently the aborts of concurrent writes might significantly affect performance. Long transactions also suffer under the first committer wins strategy. In contrast, the SER protocol aborts readers and this can easily deteriorate the performance of queries. To avoid such cases, we propose an approach that treats queries differently than update transactions. It must be noted, however, that to be able to distinguish between queries and update transactions, transactions must be declared read-only or update in advance.

This approach uses SER for update transactions and SI for queries. This combination provides full serializability. The disadvantage is that both approaches must be implemented simultaneously leading to more administrative overhead. Both update transactions and data items must receive timestamps to be able to reconstruct snapshots for the read-only transactions, and the lock manager must be able to handle the read locks of the update transactions. However, update transactions and queries do not interfere with each other and writers are not delayed by long queries. Hence, the overhead might be justified, since a replicated database makes sense only for read intensive applications.

4.1.5 Shadow Copies vs. Deferred Writes

Disadvantages of Deferred Writes Deferring the write operations has helped to present the protocols in a rather simple form and to concentrate on the different levels of isolation. In some systems, however, deferring writes might be undesirable or unfeasible. First, it is not suitable for transactions with write/read dependencies. Read operations might depend on previous write operations of the same transaction. As an example, a transaction might first insert a couple of records in a table and then want to read them. If updates are deferred, such a transaction will not see its own changes. A second problem is constraints. A write operation might not be successful because it violates some constraints (e.g., a value of an attribute has to be positive). Constraints are usually checked as part of the execution of the write operation. Although the write lock can successfully be granted during the lock phase and the owner sends a commit message, at the time the write operation is executed the constraint might not hold and hence, the transaction will abort. This can only be avoided if the local node checks constraints during the read phase acquiring the corresponding read locks. A third issue is triggers. Write operations might fire triggers that possibly generate further updates which must also be performed in the context of the transaction. Triggers, however, are usually only detected and fired upon the execution of the operation and hence, after the lock phase. In this case, additional write locks would be acquired during the write phase. Therefore, the existence of triggers can violate the requirement that transactions are serialized according to the delivery order of their write sets. A last problem arises in relational databases using SQL-based languages. Here, the write operations `update` and `delete` can contain implicit read operations. For instance, assume an update statement like

```
UPDATE a-table SET attr1 = attr1 + 100 WHERE attr1 < 2000.
```

On the one hand, the records to be updated are only determined while executing the statement since the `where`-clause contains an implicit read operation on the table. On the other hand, all write locks must be requested before the write operation is executed and the granularity of the write lock must cover at least all qualifying locks. If the system provides some form of predicate locking like, e.g., key range locking [Moh90a, Moh90b], it could be used as the perfect locking granularity. However, if this type of locking is not provided, we have to acquire a coarser lock, for instance a lock on the table. With this, concurrency can be severely restricted.

Shadow Copies A way to avoid the disadvantages of deferred write operations we propose to execute the write operations during the local read phase using shadow copies. This approach has similarities to the

I. *Local Read Phase:*
   1. *Read operations:* Acquire local read lock for each read operation $r_i(X)$. If the transaction has a shadow copy for X execute the operation on the shadow copy, otherwise execute the operation on the valid version of the data item.
   2. *Write operations:* Acquire local RIW (read intention write) lock for each write operation $w_i(X)$. If T_i has not yet a shadow copy of X create T_i's personal shadow copy of X. Execute the write operation on the shadow copy.

III.1.b If there is a write lock on X, or all read and RIW locks on X are from transactions that have already processed their lock phase, then enqueue the lock request.

III.1.c. If there is a granted read lock or RIW lock of transaction T_j and the write set WS_j of T_j has not yet been delivered, abort T_j and grant $w_i(X)$. If WS_j has already been sent, then multicast abort message a_j (basic service).

IV. *Write Phase:* Whenever a write lock on data item X is granted
   1. If T_i is local then transform T_i's shadow copy of X into the valid version of X.
   2. If T_i is remote execute the write operation or apply the updates on the valid version of X.

Figure 4.5: Protocol changes: serializability using shadow copies (SER-S)

multiversion algorithm described in [BHG87]. The idea is to execute write operations during the read phase on shadow copies of the corresponding data items. Whenever the transaction submits a write operation on a certain data item for the first time, a shadow copy of the item is created which is only visible to the transaction. From then on, whenever the transaction wants to access the data item it accesses its shadow copy. The write operations are executed immediately over these shadow copies. We call them preliminary write operations in order to distinguish them from the actual writes during the write phase. Constraints are checked upon the preliminary writes and triggers are fired and if they require further writes they are executed immediately (also over shadow copies). Also the read operations of the same transaction read the shadow copy. However, the shadow copies are not visible to concurrent transactions as long as the transaction has not committed. Instead, concurrent transactions read the original data. At the end of the transaction the updates are sent to all sites. Since the owner has already executed the operations it has the choice to send the physical updates instead of the write operations. In this case the remote sites do not need to reexecute the operations but only apply the updates to the specific data items. For the owner of a transaction, whenever the transaction has passed the lock phase and the locks are granted, the shadow copies become the valid versions of the data. As a result, the write phase on the local site is very short because it does not contain the actual operations. The protocols presented so far can be easily extended to use shadow copies instead of deferring writes.

Serializability with Shadow Copies (SER-S) To ease the comparison with the SER-D protocol, Figure 4.5 only depicts the steps of SER-S that differ from SER-D. The local read phase now executes write operations on shadow copies (phase I.2) and read operations read the shadow versions (phase I.1). During the write phase (phase IV) the local node transforms the shadow copies into the valid versions, the remote sites apply the updates.

In this protocol, concurrency control is more complex. Although shadow copies are not visible until commit time, they require a sophisticated handling of locks. As we have mentioned before, write operations in relational systems are complex operations. In particular, there might occur `update/update`, `delete/update` and `insert/insert` conflicts. Assume write operations on shadow copies would not acquire locks and there are two transactions T_1 and T_2 with the following update operations:

Lock Types	Read Lock	RIW Lock	Write Lock
Read Lock	√	√	X
RIW Lock	√	X	X
Write Lock	X	X	X

Figure 4.6: Conflict matrix when using SER-S

```
T1:UPDATE a-table set attr1 = attr1 + 1 WHERE tuple-id = 5
T2:UPDATE a-table set attr1 = attr1 + 2 WHERE tuple-id = 5
```

Both might be on the same site or on different sites and they perform the updates concurrently on shadow copies. Now assume, both send their write sets and WS_1 is delivered before WS_2. Since neither T_1 nor T_2 have locks set on the data during the read phase, first T_1's and then T_2's updates will be applied. This results in a non-serializable execution since T_2's update depends on a value it has read before T_1's update and hence, T_2 should be serialized before T_1. On the other hand, T_2's update is applied after T_1's update requiring it to be ordered after T_1. This is an example of a lost update (T_1's update is lost). The problem here is that `update` operations might have implicit read operations. For delete/update conflicts, the problem is incompatible writes. Assume T_1 deleting a data item and T_2 concurrently updating the same data item and WS_1 is delivered before WS_2. While T_2 could locally update the data item during the read phase the write phase will fail because T_1 has deleted the item. A similar problem arises if two transactions want to insert a data item and both use the same identifier for the item. While both local inserts succeed during the read phase only one can be successful during the write phase.

To avoid these problems, we use a similar approach as the multiversion 2-phase-locking scheme proposed in [BHG87]. The approach is also related to *update mode* locks [GR93]. The idea is to obtain *read-intention-write* (RIW) locks for all write operations during the read phase (read phase I.2) which are then upgraded to write locks during the lock phase. A RIW lock conflicts with other RIW locks and with write locks but not with read locks (see the conflict matrix in Figure 4.6). As a result, a transaction can perform a write operation on a shadow copy while concurrent transactions can still read the (old) version of the item. However, there is at most one transaction with a shadow copy of a data item on a certain node. By using this mechanism, the problems described above are either avoided or made visible. Conflicts between two local transactions are handled by allowing at most one RIW lock on a data item. This means that if two local transactions in their read phase want to write the same data item, one has to wait until the other has committed or aborted. Conflicts between local and remote transaction are detected during the lock phase of the remote transaction. In this case, RIW locks behave like read locks (steps III.1.b and III.1.c). If a transaction in its lock phase wants to set a write lock, it will abort all local transactions in their read or send phases with conflicting read or RIW locks.

One problem with RIW locks is that they reintroduce deadlocks. Assume transaction T_1 updates data item X and T_2 updates data item Y both holding RIW locks. If now T_1 wants to set a RIW lock on Y and T_2 wants to

I. *Local Read Phase:*
   1. *Read operations*: For each read operation $r_i(X)$, if T_i has a shadow copy of X, read the shadow copy. Else, reconstruct the version of X labeled with T_j where T_j is the transaction with the highest $TS_j(EOT)$ so that $TS_j(EOT) \leq TS_i(BOT)$.
   2. *Write operations*: For each write operation $w_i(X)$: if T_i has not yet a shadow copy of X perform first a conflict test. If the current version of X is labeled with T_j and $TS_j(EOT) > TS_i(BOT)$ then abort T_i. Else create T_i's personal shadow copy of X and execute the write operation on the shadow copy. If T_i has already a shadow copy, simply use the shadow copy.

IV. *Write Phase:* Whenever a write lock on data item X is granted
   1. If T_i is local then transform T_i's shadow copy of X into the valid version of X.
   2. If T_i is remote execute the write operation on the valid version of X.

Figure 4.7: Protocol changes: snapshot isolation using shadow copies (SI-S)

set a RIW lock on X, a deadlock will ensue. However, such a deadlock only occurs among local transactions in their read phases and therefore can be handled locally without any impact on the global execution. Note that, once a transaction is in its send phase, it will not be involved in a deadlock anymore because write locks have precedence over any other type of locks and conflicting transactions will be aborted.

Cursor Stability with Shadow Copies (CS-S) Shadow copies are introduced into the CS protocol in the same way as in the SER protocol. Write operations also acquire RIW locks that are kept until the lock phase. Upon RIW/write conflicts the local transaction holding the RIW lock is aborted. Hence, RIW locks behave like long read locks.

Snapshot Isolation with Shadow Copies (SI-S) Figure 4.7 depicts those steps of the SI-S algorithm using shadow copies that differ from the SI-D algorithm. Again, write operations are executed during the read phase on shadow copies (step I.2). The algorithm performs a preliminary version check the first time a transactions wants to update a data item. If a conflict is detected the transaction is aborted immediately. Note that the version check will be repeated during the lock phase III to detect conflicts with write operations that were performed later on. Note also that shadow copies are different copies than the copies that are reconstructed for reading purposes. They should also not be mistaken for the final versions labeled with the transaction identifiers. Shadow copies are preliminary non-visible copies and will only be transformed into the final versions when the write locks have been successfully acquired and the transaction commits (write phase IV).

It would also be possible to use RIW locks for snapshot isolation to detect conflicts not only when the shadow copy is first created but also when the preliminary write operation conflicts with subsequent write operations of other transactions. This means a transaction T would acquire a RIW lock before a preliminary write operation (I.2) and T would be aborted if another transaction wants to set a conflicting write lock while T is still in its read phase (similar to step III.1.c in Figure 4.5). Note, that it would still not be necessary for the owner of T to send an abort message to the other sites. All conflicts will be detected by the remote sites through the version check.

Discussion As discussed above, executing the write operations during the local phase avoids many of the disadvantages that can occur with deferred writes. Additionally, remote sites might be able to apply the physical updates instead of executing the entire write operation. This can lead to significant less overhead at the remote sites. In systems that do not support multiple versions, implementing the shadow copy approach might be complex. As an alternative, these systems could directly perform the write operations on valid local data copies. The only difference in this case would be, that read operations conflict with local write operations, i.e., RIW locks would not be compatible with read locks and local transactions would delay each other upon read/RIW and RIW/read conflicts.

However, executing writes during the local phase has its own drawbacks. We expect abort rates to be higher than with the deferred write approach for two reasons. The read phase – in which a transaction can potentially be aborted and where conflicts must be resolved – is now much longer. Furthermore, introducing locks also for write operations in the case of SER and CS increases the abort rates, since a transaction can now be aborted upon a read/write and a RIW/write conflict. In short, the execution of a transaction is nearly completely optimistic and the synchronization point is moved to the very end of the transaction. Only local transactions with conflicting write operations are delayed, all other conflicts lead to aborts. It should also be noted that the disadvantages of deferring writes do not necessarily apply to all database systems. Write/read dependencies might not occur often or be irrelevant. Constraints could be checked during the read phase even if the write is not executed, and there exist many systems that do not even support triggers. Furthermore, predicate locking avoids coarse locking granularity even if write operations contain implicit reads. In object oriented systems the locking granularity might not even be a problem at all. Hence, the choice of whether deferring writes or executing them during the read phase might strongly depend on the underlying system.

4.2 Correctness of the Protocols (failure-free environment)

This section contains the proofs (deadlock freedom, atomicity, 1-copy-equivalence and isolation level) of the presented algorithms in a failure free environment.

4.2.1 Serializability with Deferred Writes

We will first prove the correctness for SER-D. The proofs assume that write operations, once they have acquired the write locks, can be executed successfully.

Lemma 4.2.1 *The SER-D protocol is deadlock free.*

Proof We first show that a transaction will not deadlock after its read phase.

- A transaction that started the send phase but whose write set is not yet delivered at any node cannot be involved in a deadlock since all read locks are granted, and write locks have not yet been requested.
- Assume now $T_1 \rightarrow \ldots \rightarrow T_{i-1} \rightarrow T_i \rightarrow T_1$ to be a deadlock containing at least one transaction whose write set has been delivered at any node N. Let – without loss of generality – T_1 be this transaction. T_1 waits at N for a write lock $w_1(X)$ to be granted. Assume now that T_i – the transaction T_1 is waiting for to release the conflicting lock – is in its read or send phase when WS_1 is delivered (T_i is local to N). In this case T_i can only hold a read lock on X and would have been aborted upon the delivery of WS_1 (III.1.c). Hence, T_i is already in its write phase at N and WS_i has been delivered before WS_1. This means T_i also waits for a write lock $w_i(Y)$ to be granted (this can be on N or any other node N' in the system). Hence, we can use the same argument recursively for T_i and all other transactions in the deadlock. This means for all $T_i \rightarrow T_j$ in the cycle, WS_i has been delivered before WS_j at some node

N. Since all nodes deliver write sets in the same order, such a cycle is impossible and there cannot be such a deadlock.

Since deadlocks which only consists of transactions in their read phases cannot exist (only read locks are requested), the protocol is deadlock free. □

Theorem 4.2.1 *In a failure free environment, the SER-D protocol guarantees the atomicity of a transaction.*

Proof We have to show that if a node N commits/aborts a local update transaction T then all nodes commit/abort T. Using SER, the owner N of T always decides on commit/abort. We have to show that the remote nodes are able to obey this decision. If N aborts T during its read phase this is trivially true (the other sites do not even know about the transaction). Furthermore, an abort decision can easily be obeyed, because remote nodes do not terminate a transaction until they receive the decision message from N. The decision to commit can be obeyed because due to the lack of deadlocks, all nodes will eventually be able to grant all write locks for T and execute the operations. □

Theorem 4.2.2 *The SER-D protocol is 1-copy-serializable.*

Proof We now show that the executions at the different nodes provide indeed 1-copy-serializability. The main argument used will be the total order in which write sets are delivered at all sites. We base our proof on the basic serializability model for replicated databases and 1-copy-serializability as developed in [BHG87]. A serialization graph $SG(H)$ for the replicated data history H is a graph where the nodes correspond to committed transactions in the history. The graph contains an edge $T_i \rightarrow T_j$ if there is a node N with replica x of object X and there are conflicting operations $o_i(x)$, in T_i, and $o_j(x)$, in T_j, such that o_i is ordered before o_j in H. This means, $SG(H)$ orders all transactions that have conflicting operations on the same node. Note that we consider a read operation $r_i(X)$ to take place directly after the corresponding read lock has been acquired. Similarly, a write operation takes place when the write lock has been acquired.

A replicated data serialization graph $RDSG(H)$ is an extension of $SG(H)$ with enough edges added such that the graph orders transactions whenever they have conflicting operations on the same logical object. Formally, $RDSG(H)$ induces a write and a read order for H. The write order requires that if two transactions T_i and T_j both write the same logical object (not necessarily the same copy) then either T_i precedes T_j or the other way around. The read order requires that if T_j reads a copy x of an object X and x was last written by T_i, T_k writes any copy of X and T_i precedes T_k ($k \neq i, k \neq j$), then also T_j precedes T_k. To show that SER-D only produces 1-copy-serializable histories we have to show that each history H has an acyclic $RDSG(H)$:

Let H be a history produced by the SER-D protocol. Let $SG(H)$ be the serialization graph of H. We show that $SG(H)$ itself induces a read and a write order (mainly because of the ROWA approach), and that $SG(H)$ is acyclic. This means that for each history H, the serialization graph $SG(H)$ is an acyclic $RDSG(H)$ for H.

$SG(H)$ induces a write order:

SER-D performs always all write operations on all copies (ROWA). Therefore, if T_i and T_j have conflicting write operations on the same logical object X, then they have conflicting operations on all physical copies. Therefore $SG(H)$ contains $T_i \rightarrow T_j$ or $T_j \rightarrow T_i$.

$SG(H)$ induces a read order:

Let T_i update all copies of object X and let T_j read copy x of X that was written by T_i. This means $SG(H)$ contains $T_i \rightarrow T_j$. Let now T_k update X and T_k is ordered after T_i. Since T_k also updates the physical copy

x read by T_j, (ROWA), T_j and T_k are directly ordered to each other. Assume $T_k \rightarrow T_j$. Since $T_i \rightarrow T_k$, this would mean $T_i \rightarrow T_k \rightarrow T_j$ and T_j would read from T_k and not from T_i. This is a contradiction and hence $T_j \rightarrow T_k$.

$SG(H)$ is acyclic:

For this part of the proof we use the fact that all write sets are totally ordered, i.e., if $WS_i < WS_j$ at one site, then $WS_i < WS_j$ at all sites. We define the following partial order $<_T$ among transactions based on this total order $<$ of write sets at all sites:

- If T_i and T_j are update transactions, then $T_i <_T T_j$ iff $WS_i < WS_j$.
- For read-only transactions we introduce empty write sets. The dummy write set WS_i of a read-only transaction T_i at node N is said to be processed at the same time as its last read lock is granted. If we assume that the lock manager is not multithreaded, then WS_i is totally ordered in regard to all write sets at node N. This means
  - All update transactions and read-only transactions local on N are totally ordered.
  - Two read-only transactions initiated at different sites are not ordered if their dummy write sets are processed in between the same consecutive write sets of update transactions.

We show that if $T_i \rightarrow T_j$ in $SG(H)$, then $T_i <_T T_j$. Hence, $SG(H)$ cannot have any cycles but has dependencies only in accordance to the total order provided by the communication system, i.e., if T_i precedes T_j in $SG(H)$ then WS_i was delivered before WS_j.

An edge $T_i \rightarrow T_j$ in $SG(H)$ exists because of three different kind of conflicting operations (on the same copy): $(r_i, w_j), (w_i, w_j)$ or (w_i, r_j).

- (r_i, w_j): WS_i must have been processed before WS_j, otherwise T_i would have been aborted (step III.1.c of the protocol in Figure 4.1) and would not be in $SG(H)$. Therefore $WS_i < WS_j$.
- (w_i, w_j):
  This is only possible when $WS_i < WS_j$, because write operations are executed according to the total order write sets are delivered (step III, especially III.1.b).
- (w_i, r_j):
  T_j can only read from T_i when WS_i has been processed. Therefore, the write lock of T_i must have been granted before the read lock of T_j. Since all read operations of T_j must be performed before WS_j is sent (step II), WS_j was sent and processed after WS_i had been processed, therefore $WS_i < WS_j$.
  □

4.2.2 Serializability with Shadow Copies

As noted in Section 4.1.5 the introduction of RIW can lead to deadlocks. However, these deadlocks are completely local and only involve transactions in their read phases. Hence, they can be detected locally at one site. Global deadlocks do not exist.

Lemma 4.2.2 *Every deadlock that can occur with the SER-S protocol only involves transactions that have the same owner and are in their read phases.*

Proof The proof is similar to the proof for Lemma 4.2.1 for the SER-D protocol. The proof for Lemma 4.2.1 shows, that a transaction cannot be involved in any deadlock once it has finished its read phase. This is true for the SER-S protocol with exactly the same reasoning. Hence, every deadlock consists only of transactions in their read phases. Different to SER-D there can exist such deadlocks since transactions can acquire RIW locks. Since transactions are executed completely locally on their owner node during their read phase all transactions involved in a deadlock must have the same owner. □

Theorem 4.2.3 *In a failure free environment, the SER-S protocol guarantees the atomicity of a transaction.*

Proof The proof is identical to the proof for SER-D. □

Theorem 4.2.4 *The SER-S protocol is 1-copy-serializable.*

Proof The serializability proof is similar to the one for SER-D although write operations are executed on shadow copies during the read phase. The proof is similar because shadow copies do not have any influence regarding concurrent transactions and can therefore be ignored. Only the final write during the write phase counts. To illustrate this we again look at the different types of conflicts:

- (r_i, w_j):
  This case is only possible when WS_i has been delivered before WS_j otherwise T_i would be aborted. The preliminary write operation of T_j can in fact take place before, at the same time, or after the execution of T_i's read operation. It has, however, in no case an influence on T_i because T_i does not read the shadow copy produced by T_j's write operation.
- This is only possible when $WS_i < WS_j$, because otherwise T_i would be aborted because of a RIW/write conflict.
- (w_i, r_j):
  T_j only reads from T_i when T_i has committed, i.e., after WS_i has been delivered and executed. This means the preliminary execution of the write operation during T_i's read phase plays no role.

In fact, the only one who actually sees a shadow copy of a transaction T_i is T_i itself, namely when it first performs a write operation and then a read operation on the same data item. However, this is not a conflict, and hence, has no impact on the composition of the serialization graph. □

4.2.3 Cursor Stability

The CS-D and CS-S protocols behave in regard to deadlocks and atomicity as their SER counterparts.

Theorem 4.2.5 *The CS-D protocol is deadlock free. The CS-S protocol only allows transactions to be involved in deadlocks during their read phases. In case of a failure free environment, both CS-D and CS-S guarantee the atomicity of a transaction.*

Proof The proofs are identical to the corresponding proofs of the SER protocols. □

Theorem 4.2.6 *CS-D and CS-S provide 1-copy-equivalence and regarding the level of isolation, they avoid the phenomena $P1 - P2$, but $P3 - P5$ may occur.*

Proof The proofs for CS-D and CS-S are identical. CS does not provide serializability. Hence, we cannot apply the combined 1-copy-serializability proof using a replicated data serialization graph. 1-copy-equivalence itself means that the multiple copies of an object appear as one logical object. This is true because of several reasons. First, we use a read-only/write-all approach, i.e. transactions perform their updates on all copies. Second, transaction atomicity guarantees that all sites commit the updates of exactly the same transactions. Third, using the total order guarantees that all these updates are executed in the same order at all sites. Hence, all copies of an object have the same value if we look at them at the same logical time (for example, let T_i and T_j be two transactions updating object X, WS_i is delivered before WS_j and we look at each node at the time the lock for the copy of X is granted to T_j. At that time each copy of X has the value that was written by T_i).

The phenomena P1 and P2 are avoided because of the following reasons:

P1 Dirty read: $w_1(X_i)\dots r_2(X_i)\dots(c_1$ or $a_1)$ is not possible because updates become visible only after commit of a transaction. Read operations, whether obtaining short or long locks, do not see shadow copies of concurrent transactions but only committed values.

P2 Lost update: $r_1(X_i)\dots w_2(X_i)\dots w_1(X_i)\dots c_1$ is not possible because T_1 should obtain a long read lock if it wants to write X (according to step I.1 in Figure 4.3) . Therefore, it will be aborted when WS_2 is processed.

CS does not avoid the other phenomena, i.e., there exist executions in which these phenomena occur. This is easy to see since most read locks are released immediately after the operation allowing other transactions to access the data. □

4.2.4 Snapshot Isolation

Lemma 4.2.3 *The SI-D and SI-S protocols are deadlock free.*

Proof Transactions only acquire write locks and all at the same time. Hence, deadlocks cannot occur. □

Theorem 4.2.7 *In a failure free environment, SI-D and SI-S guarantee the atomicity of a transaction.*

Proof The proof is the same for SI-D and SI-S. With SI the local node N only sends the write set WS_i and all nodes decide on their own on the outcome of the transaction. To show the atomicity of a transaction we have to show that all nodes make the same decision. This is done by induction on the sequence of write sets that arrive (this sequence is the same at all nodes). Hence, we use the total order to prove the atomicity of transactions.

Assume an initial transaction T_0 with $TS_0(EOT) = 0$. All objects are labeled with T_0. Now assume WS_i is the first write set to be delivered. The $BOT(T_i)$ must be 0. Therefore all nodes will decide on commit and perform all operations. Now assume, already $n-1$ write sets have been delivered and all nodes have always behaved the same, i.e., have committed and aborted exactly the same transactions. Therefore, on all nodes there was the same series of committed transactions that updated exactly the same objects in the same order. Since these transactions have the same EOT timestamps at all nodes, the versions of the objects are exactly the same at all nodes when all these transactions have terminated. Hence, when WS_i is the n'th write set to be delivered, $\forall nodes, \forall w_i(X) \in WS_i$ the version check in step III of Figure 4.4 will have the same outcome. Although at the time WS_i is processed not all of the preceding transactions might have committed, the check is always against the last version. This version might already exist if all preceding transactions that updated this object have already committed and there is no granted lock on the object (step III.1 of the protocol), or the version will be created by the transaction that is the last still active preceding one to update the object (III.2 of the protocol). □

Theorem 4.2.8 *SI-D and SI-S provide 1-copy-equivalence and avoid phenomena $P1-P4$, but $P5$ may occur.*

Proof 1-copy-equivalence is guaranteed for the same reasons as it is guaranteed for the CS protocols. Phenomena $P1-P4$ are avoided due to the following reasons:

P1 Dirty read: $w_1(X_i)\dots r_2(X_i)\dots(c_1$ or $a_1)$ is not possible since updates become visible only after commit of a transaction. In our algorithm, $TS_2(BOT)$ is lower than $TS_1(EOT)$ and therefore T_2 will not read from T_1 (step I.1 of the protocol in Figure 4.4).

P2 Lost update: $r_1(X_i) \ldots w_2(X_i) \ldots w_1(X_i) \ldots c_1$ is not possible. $TS_2(EOT)$ is bigger than $TS_1(BOT)$ but smaller than $TS_1(EOT)$. Therefore, in the moment in which $w_1(X_i)$ is requested T_1 will be aborted (step III of the protocol). In the case of SI-S, if T_2 has committed before T_1 executed the first preliminary operation on X_i, the conflict will already be detected during the read phase (step I.2).

P3 Non-repeatable read: $r_{11}(X_i) \ldots w_2(X_i) \ldots c_2 \ldots r_{12}(X_i)$ is not possible. $TS_1(BOT)$ is lower than $TS_2(EOT)$ and therefore T_1 will not read from T_2 at its second read but reconstruct the older version (step I.1 of the protocol).

P4 Read skew: $r_1(X_i) \ldots w_2(X_i) \ldots w_2(Y_j) \ldots c_2 \ldots r_1(Y_j)$ is not possible since a transaction only reads data-versions created before the transaction started or its own updates (step I.1 of the protocol).

P5 Write skew: $r_1(X_i) \ldots r_2(Y_j) \ldots w_1(Y_j) \ldots w_2(X_i)$ is possible since the two write operations do not conflict and read/write conflicts are not considered. □

4.2.5 Hybrid Protocol

The proofs of correctness for the hybrid protocol can be directly derived from the proofs for the underlying protocols. In this protocol, queries do not have any influence on deadlocks since queries do not acquire any locks. Hence, deadlocks cannot occur if update transactions use the SER-D protocol. They can occur, if update transactions use the SER-S protocol. Furthermore, atomicity is directly given by the SER protocol.

Theorem 4.2.9 *The Hybrid protocol using SER for update transactions and SI for queries is 1-copy-serializable.*

Proof All histories containing only update transactions are 1-copy-serializable due to the SER protocol. This is true despite concurrent queries since queries do not acquire any locks and never block update transactions and hence, do not have any influence on the execution of update transactions. Furthermore, we have shown above that any execution with the SER protocol is equivalent to a serial execution in which all update transactions are ordered according to the total order delivery.

Let T_i now be a query. According to the SI protocol the BOT timestamp $TS_i(BOT)$ is set to the highest sequence number of a message WS_j so that transaction T_j and all transactions whose WS have lower sequence numbers than WS_j have terminated. This means, we can order T_i directly after T_j and before the transaction succeeding T_j in the total order. We have to show the following:

- There is no conflict between T_i and an update transaction T_k with $TS_k(EOT) > TS_j(EOT)$, that would require that T_i is ordered after T_k, i.e., there is no conflict (w_k, r_i). This is true because T_i will not read any value written by a transaction with higher sequence number than WS_j according to the SI protocol (step I.1).
- There is no conflict between an update transactions T_k with $TS_k(EOT) \leq TS_j(EOT)$ that would require that T_i is ordered before T_k, i.e., there does not exist a conflict (r_i, w_k). This is true because if T_i conflicts with T_k on data item X it will either read the version of X written by T_k or a version written by a transaction T_l with $TS_k(EOT) < TS_l(EOT)$, but never a version written by a transaction T_l with $TS_l(EOT) < TS_k(EOT)$ according to the SI protocol (step I.1). □

4.3 Protocols with different Levels of Fault-Tolerance

So far we have presented a suite of protocols providing different degrees of isolation. This section refines these protocols to provide different degrees of fault-tolerance. In the following, we do not distinguish

Protocol Name \ Message Type	Write Set	Decision Message
SI–UR	UR	–
SI–R	R	–
SER–UR / CS–UR / Hybrid–UR	UR	UR
SER–URR / CS–URR / Hybrid–URR	UR	R
SER–R / CS–R / Hybrid–R	R	R

Table 4.1: Combinations of levels of isolation and degrees of reliability

between the deferred write and shadow copy alternatives since their differences do not have any impact on fault-tolerance issues.

In Section 3.2.2 we described two degrees of reliability which are usually offered by group communication systems for the delivery of messages: uniform reliable (UR) and reliable (R) message delivery. Our approach is to combine these two degrees of reliability with the protocols presented so far. Each of the SER/CS/SI/Hybrid protocols multicasts one or two messages per transaction. For each of these messages we have the choice to send the message with either uniform reliable message delivery or with reliable message delivery. Hence, we obtain for each level of isolation two or three protocols which differ in the guarantees they provide in case of failures.

- *SI protocol*: The SI protocol only sends one message per transaction (the write set). It can either be sent with uniform reliable delivery (SI-UR) or reliable delivery (SI-R).
- *SER/CS/Hybrid protocols*: The SER/CS/Hybrid protocols send two messages, namely the write set and the decision message. Both of these messages can be sent with uniform reliable delivery (SER-UR/CS-UR/Hybrid-UR), both can be sent with reliable delivery (SER-R/CS-R/Hybrid-R), or the write can be sent with uniform reliable delivery while the decision message uses reliable delivery (SER-URR/CS-URR/Hybrid-URR). To send the write set with reliable delivery and the decision message with uniform reliable delivery does not have a different effect than sending both with reliable delivery and will therefore not be discussed.

Table 4.1 summarizes the possible combinations. For each of them we will discuss what needs to be done when failures occur and which atomicity guarantees can be provided. As described in Section 3.2.2, node failures lead to a new view at the remaining nodes. Hence, in the following we will look at run R_i at each node N, i.e., from installing view V_i until installing a new primary view V_{i+1} (N is V_i-available) or the failure of N. The extension to an entire execution is straightforward. In particular, we discuss what the V_i-available nodes have to do when the new primary view V_{i+1} is installed. Furthermore, we look which transactions will be committed at the V_i-available and the faulty sites.

Based on the properties of view synchrony we will show that independently of the degree of reliability, transaction atomicity and data consistency are guaranteed on all V_i-available sites. This means, all V_i-available sites commit exactly the same transactions in the same order. Furthermore, we show that using uniform reliable delivery, atomicity is guaranteed in the entire system. This means that the set of transactions committed respectively aborted at a failed node is a subset of the transactions committed respectively aborted at available nodes with the same serialization order. Furthermore, we show what kind of atomicity violations can occur on the faulty sites when we use reliable message delivery. Our proofs will refer to the characteristics of the virtual synchrony and uniform reliable respectively reliable message delivery as defined in Section 3.2.2.

4.3.1 Snapshot Isolation

For nodes that change from primary view V_i to primary view V_{i+1}, there is no difference between SI-R and SI-UR. For both types of delivery, when a node N is in view V_i and receives view change message V_{i+1}, it simply continues processing transactions.

Let $\mathcal{T}$ be the set of transactions whose write sets have been sent in view V_i.

Theorem 4.3.1 *Both the SI-R and SI-UR protocols guarantee atomicity of all transactions in $\mathcal{T}$ and data consistency on all V_i-available nodes.*

Proof Both reliable and uniform reliable delivery together with the total order of WS messages (guarantee III of virtual synchrony) and the view synchrony (guarantee I) guarantee that the database modules of V_i-available nodes receive exactly the same sequence of WS messages before receiving view change V_{i+1}. Furthermore, liveness (II) guarantees that write sets of V_i-available nodes will always be delivered. Thus, within this group of available nodes (excluding failed nodes), we have the same delivery characteristics as we have in a system without failures. Hence, we can rely on the atomicity guarantee and 1-copy-equivalence of the SI protocol in the failure-free case. □

Uniform reliable and reliable delivery for SI differ in the set of transactions committed at failed nodes and, consequently, in what must be done when nodes recover.

Snapshot Isolation with Reliable Delivery (SI-R) If SI-R is used, a failed node (crashed or partitioned) might commit transactions that the available nodes do not commit. This can happen since a failed site might have delivered a write set before failing but none of the other sites deliver this write set. Depending on which total order protocol is implemented it might also have transactions ordered differently if the write sets were delivered in a different order than at the available sites. For instance, using the Totem [MMSA$^+$96] protocol, assume the communication module of node N is the owner of the token and up so far it has delivered all messages up to a sequence number i to the database module. Now the database module wants to multicast the write set WS of a transaction T. The communication module assigns WS the sequence number $i + 1$ and broadcasts it to the other nodes. At the same time it can deliver WS back to its local database module immediately (because it has already delivered all messages up to i). Now assume the database executes T and commits it. Furthermore assume the message itself gets lost on the network. Eventually, the other sites would detect this loss. However, before they are able to do so, node N fails and is therefore not able to retransmit the write set. Hence, the other sites will never receive WS, i.e., they ignore T which is identical to an abort. Upon recovery, the failed node needs to reconciliate its database with that of the working nodes to compensate the updates done by the wrongly committed transactions.

Snapshot Isolation with Uniform Reliable Delivery (SI-UR) Using SI-UR, the set of transactions committed respectively aborted at a failed node is a subset of transactions committed respectively aborted at available nodes. Intuitively, this is true since a node failing during V_i delivers a prefix of the sequence of messages delivered by a V_i-available node. Thus, the behavior of an available node and a failed node is the same until the failure occurs and the failed node simply stops or moves to a minority view.

Theorem 4.3.2 *The SI-UR protocol guarantees the atomicity of all transactions on all sites.*

Proof To prove the theorem, we look at a node N which fails during view V_i and at the different states a transaction T_j on node N might be in when N fails. For each case we show that N does not contradict the decision that is made by the V_i-available (or short: available) nodes or by any other node that fails during V_i. This means that when N has committed/aborted T_j the available nodes do the same. On other nodes failing during V_i, T_j was either still active or terminated in the same way. When T_j was still active on N then any other site only commits the transaction when this decision is correct (N would have been able to commit it too, if it had not failed). Otherwise they abort it.

Consider a transaction T_j at a failed node N:

1. T_j is in its read phase (T_j is local): In this case none of the other nodes knows of T_j. The transaction has no effect neither on N nor on any other node. Hence, T_j can be considered aborted on all sites.
2. T_j is in its send phase (T_j is local and WS_j has been sent in view V_i but not yet received by N): Reliable multicast (and hence uniform reliable multicast) guarantees, that all available nodes or none of them will receive T_j's write set (guarantee III of virtual synchrony). In the latter case, uniform reliable multicast guarantees, that no other node N' failing during V_i has received WS_j. Hence, if it is not delivered at any site the transaction is considered aborted. Otherwise, all sites that receive WS_j have received the same sequence of messages up to WS_j and hence decide on the same outcome of T_j (according to Theorem 4.2.7).
3. T_j, remote or local, is in its lock or write phase: This means that WS_j has already been delivered at N before or during view V_i. Hence, according to the uniform reliable delivery, WS_j will be delivered at all available sites (note that reliable delivery does not provide this guarantee) and possibly at some other nodes failing during V_i. Again, on all these sites (including N) the same sequence of messages is received before WS_j is received and the version check will have the same results (see Theorem 4.2.7). Hence, all can decide on the same outcome (although N will not complete T_j before it fails).
4. Finally, a transaction T_j, local or remote, was committed/aborted on N when N failed: This means WS_j is delivered on N and due to the uniform reliable message delivery, all available sites (and possible some sites failing during V_i) will receive WS_j. Hence, the same holds as in the previous case.

This together with Theorem 4.3.1 provides the atomicity of all transactions on all sites. □

4.3.2 Serializability, Cursor Stability and Hybrid

Similar to SI, for the SER, CS and Hybrid protocols there is no difference on available nodes between reliable and uniform reliable delivery. Unlike SI, however, only the owner of a transaction can decide on the outcome of the transaction. Therefore, a failed node can leave *in-doubt* transactions in the system:

Definition 4.3.1 (in-doubt transactions) *Let N be a node failing during view V_i and T a transaction invoked at node N whose write set WS has been sent before or in V_i. T is **in-doubt in** V_i if the V_i-available nodes receive WS but not the corresponding decision message (commit/abort) before installing view V_{i+1}.*

From here we can derive the set of transactions for which the outcome is determined before view V_{i+1}. Let $\mathcal{T}_1$ be the set of transactions which are in-doubt in V_i. $\mathcal{T}_2$ is the set of transactions that have been invoked at a V_i-available node and both write set and decision message have been sent before or in view V_i. Finally, $\mathcal{T}_3$ is the set of transactions that have been invoked at a node failing during V_i and both write set and decision message have been received by the V_i-available nodes before installing V_{i+1}. Let $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_3$.

Note that there are two types of transactions that are not in $\mathcal{T}$. The first type is transactions of available nodes where the decision message has not been sent in V_i. For this group, the decision will simply be made in the next view. The second group is the transactions invoked at a node failing during V_i where the write set was not received at any available site. In the following we refer to this group as $\mathcal{G}$.

Theorem 4.3.3 *The SER, CS and Hybrid protocols either with reliable or with uniform reliable delivery, guarantee transaction atomicity and data consistency of all transactions in $\mathcal{T}$ on all V_i-available nodes.*

Proof For any SER/CS/Hybrid-R, SER/CS/Hybrid-UR or SER/CS/Hybrid-URR, all sites available during V_i receive exactly the same messages (and the write sets in the same order) before installing view change V_{i+1}. Hence, they all have exactly the same set of in-doubt transactions and the same set of transactions for which they have received both the write set and the decision message. If they all decide to abort in-doubt transactions, they have the same behavior as a system without failures. Thus, atomicity and 1-copy-equivalence are provided on all available nodes. $\square$.

As with SI, the reliability of message delivery has an impact on the set of transactions committed at failed nodes and hence, determines whether consistency and transaction atomicity is also guaranteed on failed nodes.

Serializability, Cursor Stability and Hybrid with Reliable Delivery (SER-R, CS-R, Hybrid-R) Using the SER/CS/Hybrid-R protocols, a node could commit a transaction from group $\mathcal{G}$ (transactions invoked at a node failing during V_i where the write set was not received at any available site) locally before it is seen by any other node. Consequently, failed nodes may have committed transactions that are non-existing at the other sites and must be reconciled upon recovery. Hence, transaction atomicity cannot be guaranteed on all nodes. Furthermore, failed nodes might have delivered write sets in a different order than the available sites and hence, committed transactions in different order.

Transactions that are in-doubt at the available sites can either be committed, active or aborted at their failed owners. Since the failed site has to reconcile its database in any case, aborting in-doubt transactions seems the adequate solution.

Serializability, Cursor Stability and Hybrid with Uniform Reliable Delivery (SER-UR, CS-UR, Hybrid-UR) In contrast, using the SER/CS/Hybrid-UR protocols (both messages use uniform reliable delivery), consistency and transaction atomicity is guaranteed in the entire system (including failed nodes). Upon reception of the view change, all available nodes will have the same in-doubt transactions which are safe to abort. These transactions are either active or aborted on the failed nodes but never committed, since uniform delivery guarantees that all or no node receive the commit. Furthermore, transactions from $\mathcal{G}$ (that are not visible at available nodes) cannot be committed at failed nodes, since this is only possible when both write set and commit message are delivered. Similarly, failed nodes cannot have delivered write sets in reverse order. The requirement of uniform reliability does not apply to abort messages, i.e., they can be sent with the reliable service, since the default decision for in-doubt cases is to abort. This approach is non-blocking since all available nodes decide consistently to abort all in-doubt transactions and the new view can correctly continue processing.

Theorem 4.3.4 *The SER-UR, CS-UR and Hybrid-UR protocols aborting all in-doubt transactions after a view change guarantee the atomicity of all transactions on all sites.*

Proof Consider a transaction T_j at a node N failing during V_i:

1. T_j is in its read phase: As with SI, this transaction is considered aborted.
2. T_j is in its send phase: T_j is still active at N and N has not yet decided whether to commit or abort T_j. Since N is the only one who can decide to commit since N is the owner (this is different to the SI protocol) no site may commit the transaction. According to the semantics of the reliable/uniform reliable multicast, all or none of the V_i-available nodes will deliver T_j's write set (guarantee III of virtual synchrony). If it is not delivered at any site the transaction is considered aborted. If it is delivered, it is an in-doubt transaction at all sites (N has not yet sent a decision message) and all available sites will abort it. Hence, they make the correct decision. Regarding other nodes failing during V_i, they either have not even received WS_j or WS_j was received and the transaction active but never committed when the node failed.
3. T_j is a local transaction of N, the write set of T_j was delivered at N but N has not yet sent the decision message: Again, this forbids that the transaction is committed at any site. Since WS_j was sent with uniform reliable message delivery, it will be delivered at all V_i-available sites and possibly at some other sites failing during V_i (guarantee III.2). On the latter, T_j will still be active when they fail. At the V_i-available sites, T_j will be an in-doubt transaction and be aborted.
4. T_j has processed its lock phase and submitted the commit message to the communication module. However, N fails before any V_i-available node physically receives the commit message: In this situation, all available sites have delivered WS_j but none of them delivers c_j. T_j is an in-doubt transaction and will be aborted. We have to show that T_j was still active on any failed node (including N). This is the case since the database module at each node waits to commit until the communication module delivers the commit message (step V of the SER/CS/Hybrid protocols). However, due to the uniform reliable delivery of the commit message, the communication module only delivers it when it is guaranteed that all sites will deliver the message. Since this is not the case here, the transaction is still pending at all failed nodes.
5. T_j has submitted the abort message (any time after sending WS_i and before the lock phase) to the communication module: Since the abort message is only sent with reliable delivery, the communication module delivers the message back to the database module immediately and does not wait until it is guaranteed that the other nodes will deliver the message. Hence, N has either aborted T_j or T_j was still active on N at the time of the failure. Since a transaction is only committed when a commit message is received, no other node commits T_j. At any node N', T_j was either active at the time N' fails, completely unknown (N' has never received the write set) or aborted (the abort message was received or T_j was in-doubt).
6. T_j, being local or remote, was committed at N before N failed: This can only happen when both write set and commit message were delivered. However, the messages are only delivered when it is guaranteed that they will be delivered at all available sites. Hence, T_j is no in-doubt transaction at the available sites and will commit. Furthermore, the uniform reliable delivery of the write set excludes the scenario where all nodes of the remaining group deliver the commit message but not the write set or where write sets are delivered in reverse order at different sites leading to different commit orders.
7. T_j, being remote, was active at N when N failed: T_j is the local transaction of either another node failing during V_i, then it is covered by the other cases, or it is the local transaction of an V_i-available node which will decide correctly on the outcome of the transaction.

We have shown that a failed node aborts/commits exactly the same transactions in the same serialization order as the available nodes until it fails. After the node failure all available nodes have the same in-doubt transactions which they can safely abort. This together with Theorem 4.3.3 guarantees the consistency and atomicity of all transaction on all sites. □.

Serializability, Cursor Stability and Hybrid with a combination of Uniform Reliable and Reliable Delivery (SER-URR, CS-URR, Hybrid-URR) Some of the overhead of uniform delivery can be avoided by risking not being able to reach a decision about in-doubt transactions. This is the case if *the write set WS is sent using uniform reliable delivery*, but *both commit and abort messages are sent using reliable delivery*. After the database module receives a view change excluding node N, it will decide to block all the in-doubt transactions of N and then continue processing transactions. A transaction that is in-doubt at the available nodes can be active, committed or aborted at the failed node (as it was for the SER/CS/Hybrid-R protocols). Hence, to achieve transaction atomicity on all nodes, in-doubt transactions must be blocked until the failed node recovers. For all other transactions, transaction atomicity can be guaranteed with the same arguments as for the SER/CS/Hybrid-UR protocols. Blocking transactions is similar to the blocking behavior of the 2-phase-commit protocol. Note that it does not imply that the nodes will be completely unable to process transactions. Transactions that do not conflict with the blocked transactions can be executed as usual. Only transactions that conflict with the blocked transactions must wait. In practice, it might also be reasonable in this case to abort these transactions but recovery needs to consider transactions committed at failed nodes and aborted elsewhere.

Theorem 4.3.5 *The SER-URR, CS-URR and Hybrid-URR protocols blocking all in-doubt transactions after a view change guarantee the atomicity of all transactions on all sites.*

Proof The proof is the same as for the previous theorem except the case which forces to block in-doubt transactions. This is case 4 when T_j has processed its lock phase and submitted the decision message to the communication module but N fails before any available node physically receives the decision message. Since both commit and abort are only sent with reliable delivery, the database does not wait until it is guaranteed that the other nodes will receive the message. Hence, T_j can be still active, committed or aborted at the time of N's failure. The uniform reliable delivery of WS_j guarantees that the V_i-available nodes have received WS_j and do not ignore T_j. However, since they do not know the outcome of T_j on the failed nodes they must block it in order to guarantee atomicity on all sites. □.

4.3.3 Further Comments

We would like to comment on two issues that need further clarification. First, we discuss the proofs in regard to the actual time at which events happen in the system. Second, we discuss the question of how inter-transaction dependency is considered correctly in the case of failures.

Asynchrony between Communication Module and Database System We would like to note that we have equated the delivery of a message with the processing of the corresponding transaction which is not exactly the case in a real system. As an example (see also Figure 4.8), assume a V_i-available node N and the communication module of N delivers a write set message WS of transaction T very late during view V_i to the database module. For example, WS is delivered during the view change protocol just before the new view V_{i+1} is installed. We have shown that N will decide about the outcome of T and terminate T exactly

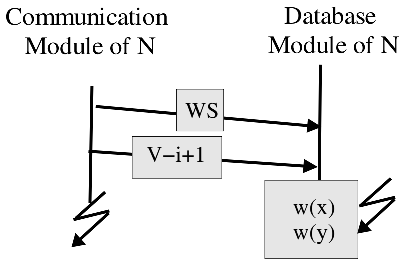

Figure 4.8: Example of asynchrony between communication module and database system

in the same way as the other V_i-available sites. Assume now the outcome decision is commit. However, N will not instantaneously commit T at the time of the delivery but some time later when the write phase has been processed. Due to this asynchrony of events (time of view change determined by the communication module and time of commit determined by the database system) N might fail after the installation of V_{i+1} but before it actually commits T. This asynchrony, however, does not change anything conceptually in the correctness of the statements regarding atomicity. Hence, it has been ignored to keep the proofs and arguments understandable. Upon recovery, care has to be taken that N will also apply the updates of T. Since T's commit is not recorded in N's log, N has to receive T's updates from a peer node (see Chapter 8).

Orphan Transactions As mentioned in Section 3.2.3 node failures might leave orphan messages in the system. These messages, although received by the available sites cannot be delivered because preceding messages on which the orphan might depend on are missing. The question is whether there also exist something like "orphan transactions". In general, a transaction depends on another transaction through the reads-from-dependency. That is a transaction T_i depends on transaction T_j if T_i has read a value that T_j has written. In our replicated system, this means that each site executing T_i must also execute T_j and serialize T_j before T_i. In a failure-free environment, this is always the case. If T_i reads from T_j at node N, N must have delivered WS_j before or while T_i is in its read phase. Hence, WS_j is delivered before WS_i at all sites and all sites apply the updates of both transactions in the same correct order. However, once failure occur, one has to be more careful. Figure 4.9 shows an example using the SI-R protocol. Assume a group of five nodes $N_1, N_2 \ldots, N_5$. N_1 sends a write set WS_1 which is received by N_1 and N_2 but not by the rest of the system. At N_2, WS_1 is delivered, T_1 executed and committed. A new transaction T_2 is started and reads one of the values written by T_1. After the read phase, N_2 sends WS_2 and WS_2 is received at all sites. However, the communication modules of N_3, N_4 and N_5 will not yet deliver WS_2 because they still miss WS_1. Before WS_1 can be resent, both N_1 and N_2 fail. As a result, although N_3, N_4 and N_5 received WS_2, T_2 may not be executed since T_1 is missing.

Note that orphans are always transactions of failed nodes. A transaction T of an V_i-available node N can never be an orphan, since N itself must have received the write sets of the transactions T has read from, and hence, all V_i-available nodes will receive them.

The correct treatment of orphans can automatically be guaranteed by the communication system if the total order service obeys causal dependencies. In this case the communication system will detect during the view change protocol that WS_2 is an orphan causally depending on the missing message WS_1, and hence not deliver WS_2 but discard it.

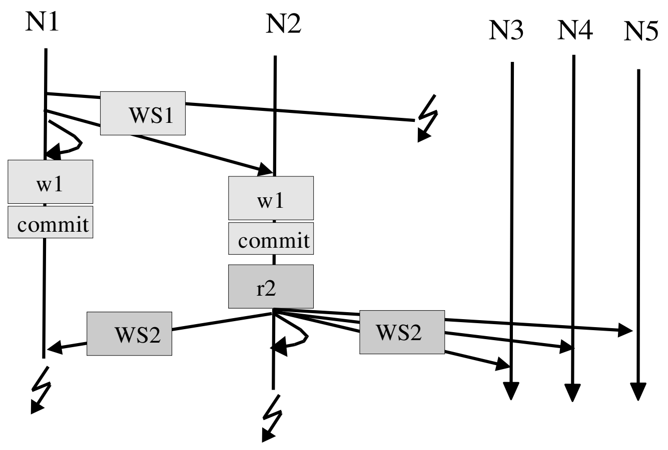

Figure 4.9: Example of orphan transactions

4.4 Discussion

4.4.1 Summary

This chapter has introduced a family of protocols that vary in three parameters. First, they implement different levels of isolation (serializability, cursor stability, and snapshot isolation). Second, they differ in how and when write operations are executed (deferred or on shadow copies). Third, they differ in their level of fault-tolerance (using uniform reliable or reliable message delivery).

The next chapter will compare these algorithms from a performance point of view. This will show the relative performance differences of the protocols under various configurations and workloads and will make suggestions which protocol to use under which circumstances. What we can expect is that the fully serializable, fully fault-tolerant protocol performs well when conflict rates are low and communication fast. Lowering the level of isolation will be necessary at high conflict rates. Lowering the level of fault-tolerance will increase performance when communication is slow. With this, our family of protocols can be seen as an attempt to provide *efficient replica control under various conditions*.

But performance is not the only criteria when the question arises which protocol to use. The feasibility of a protocol also depends on how easy it is to integrate into an existing system. We will discuss this issue in detail in Chapter 6. For instance, the choice of the level of isolation depends on the underlying concurrency control mechanism of a given database system. If the database system is based on 2-phase locking the implementation effort for integrating the SI protocol might be simply too high to be considered practicable. Deferring writes or performing them during the read phase has similar constraints. From a purely performance point of view executing writes during the read phase will suffer from more aborts but achieve more throughput if remote sites only have to apply the physical updates. In practice, the importance of including triggers and supporting write/read dependencies will favor to execute writes immediately. Also the question of fault-tolerance is not only performance related. One might want to use uniform delivery for important transactions and reliable delivery for less important transactions. However, in this case, the group communication system must offer the possibility to choose the degree of reliability individually for each message – which some of the existing systems do not provide. Hence, in this regard, flexibility is determined by the choice of the group communication system. In view of these restrictions our family of protocols can be seen as an attempt to provide *feasible replica control for various types of database systems*. With this, we do not only refer to the protocols presented in this chapter. In regard to concurrency control, for instance, we believe that the basic mechanisms of our replica control approach can be combined with many more concurrency control mechanisms.

4.4.2 Comparison with Related Work

The model we use differs significantly from quorum-based protocols where individual messages are sent for each single update, transaction ordering is provided by distributed 2-phase-locking and atomicity is guaranteed by executing a 2-phase commit protocol. It is also quite different to the epidemic approach proposed in [AES97]. Whereas we simply use the total order "off-the-shelf" as it is provided in any group communication system, [AES97] enhances the causal order of the communication system to ensure serializability. As a result, their approach needs a 2-phase-commit and requires reads to be made visible at all sites. Stanoi et. al [SAE98] exploit similarly to us different multicast ordering semantics to support the serialization of transactions. Still, their solutions require some form of 2-phase-commit. In some of their protocols, their 2-phase-commit is tightly integrated into the semantics of the multicast primitives, which makes it impossible to decouple the database system from the underlying group communication system.

The combination of eager and lazy replication proposed in [BK97] and [ABKW98] has certain similarities with our solution. In both approaches, transactions are first executed locally at a single site and communication takes place before the transaction commits in order to determine the serialization order. In both cases, the local site can commit a transaction before the other sites have applied the updates. In our approach, the local site can commit the transaction once the position in the total order is determined and relies on the fact that the remote sites will use the same total order as serialization order; in [BK97] and [ABKW98] updates are sent in any case only after the commit. Neither approach needs a two-phase-commit. However, there are also some significant differences. [BK97] and [ABKW98] send "serialization messages" and use distributed locking or a global serialization graph to determine the serialization order. Updates are sent in a separate message. We use the delivery order of the write set messages to determine the serialization order. Unlike our approach, two of the mechanisms proposed by [BK97] and [ABKW98] exchange serialization messages for each operation. The third algorithm is more similar to ours in that it first executes the transaction locally and only checks at commit time whether the global execution is serializable.

Parallel to this work, several suggestions [AAES97, HAA99, PGS97, PGS99] have been made to implement eager replication using group communication systems. The protocols proposed in [AAES97] and analyzed in [HAA99] can be seen as a motivation for the work presented in this thesis. However, their approach is more theoretical, and the presented algorithms are rather simplistic and have high communication and execution costs. By analyzing their protocols and traditional replication protocols we were able to extract the key issues that are necessary to provide efficient and correct database replication.

The work in [PGS97, PGS98, PGS99, Ped99] is similar to ours in that it also first executes a transaction locally and then sends it to all sites using a total order multicast to serialize transactions. Their approach to minimize conflicts and abort rates is to reorder transactions whenever appropriate. This means whenever the fact that the serialization order has to follow the total order leads to an abort each node tries to reorder the transactions so that the abort can be avoided. Since all sites follow the same reorder algorithm and behave deterministically all sites will have the same serialization order. Their protocol differs from ours in that they require the read set to be sent to all sites to detect conflicts. Furthermore, they do not explore the possibilities of lower levels of isolation and do not discuss fault-tolerance. In [PG97], the authors introduce the concept of different degrees of "safety" for transactions that are similar to different levels of isolation.

5 Protocol Evaluation

In order to evaluate the performance of the proposed protocols we have developed a detailed simulation tool and conducted an extensive set of simulation experiments analyzing the behavior of the protocols with respect to various parameters. This section will describe the simulation system, specify the parameters and discuss a comprehensive summary showing the general behavior and most relevant differences between the protocols. First, we will compare the deferred writes and the shadow copy approach. Then, we will analyze the behavior for different communication costs, for queries (read-only transactions) and when the system size increases. The results also include a comparison with standard distributed 2-phase locking. These performance results will allow us to draw conclusions which protocol should be used for specific configurations and workloads.

5.1 Simulation Model

The architecture of the developed simulation model [Ple97] captures the most important components of a replicated database system using similar techniques to those applied in other studies of concurrency control [ACL87, CL89, GHR97]. The simulation system is implemented using the CSIM simulation library [Sch86, Mes99] and the programming language C++. The source code consists of around 7500 lines. The general architecture corresponds to the one in Figure 3.1 of Chapter 3. Each site consists of a communication manager, a transaction manager, a data manager and concurrency control structures. Table 5.1 summarizes the simulation parameters.

5.1.1 Data and Hardware

The database is modeled as a collection of *DBSize* objects where each of the sites stores copies of all objects. Each object has the same probability to be accessed. There are *NumSites* sites in the system and each of the sites consists of one processor for transaction processing and one for communication. We use a dedicated processor for communication in order to differentiate between transaction processing and communication overhead. Furthermore, *NumDisks* disks are attached to each node. Access to processor and disk resources is competitive, i.e., each access to a resource consumes time in which the resource cannot serve other requests. Each resource has its own request queue which is processed in FCFS order. When a node wants to perform an I/O, one of its disks is chosen randomly assuming an optimal distribution of the data across the disks. The execution time of simple read and write operations on objects of the database is determined by three parameters. The *ObjectAccessTime* parameter models the CPU time (processor access time) needed to perform an operation, and *DiskAccessTime* captures the time needed to fetch an object from the disk. The *BufferHitRatio* indicates the percentage of operations on data that resides in main memory and does not require disk access.

General	*DBSize* *NumSites* *NumDisks*	Number of objects in the database Number of sites in the system Number of disks per site
Database Access	*ObjectAccessTime* *DiskAccessTime* *BufferHitRatio*	CPU object processing time Disk access time % of object accesses that do not need disk access
Communication	*SendMsgTime* *RcvMsgTime* *BasicNetworkDelay* *NetBW* *MsgLossRate*	CPU overhead to send message CPU overhead to receive message Basic delay for IP level and lower Network bandwidth Message loss rate
Transaction Type	*TransSize* *WriteAccessPerc* *RWDepPerc* *TransTypePerc* *Timeout*	Number of op. of a transaction % of write op. of the transaction type % of read op. on objects that will be written later % of the workload that belongs to this type Timeout for the 2PL algorithm
Concurrency	*InterArrival* *Mpl*	Average time between the arrival of two transactions at one node Multiprogramming level: max. number of local transactions active at one site

Table 5.1: Simulation model parameters

5.1.2 Communication Module

The communication module implements the following primitives:

- Basic multicast with reliable and uniform reliable delivery
- Total order multicast with reliable and uniform reliable delivery
- Point-to-point communication

We assume a broadcast medium where a message to a group of sites only requires a single physical message. Still, a logical multicast message involves more than one physical message (handling message loss, total order etc.). The implementation of all multicast primitives is according to the description in Section 3.2.3. For the total order multicast, we have used the Totem protocol.

Furthermore, we model a simple reliable point-to-point primitive similar to TCP/IP. Note that also for point-to-point communication we implement a reliable communication, i.e., we detect message loss in the communication system and lost messages are resent. Hence, the database system never has to take care of this.

The basic communication overhead is modeled by several parameters. Each physical message transfer has an overhead of *SendMsgTime* CPU time at the sending site and *RcvMsgTime* CPU time at the receiving site. The times may differ for different message sizes. Furthermore, we assume a time overhead of *BasicNetworkDelay* (for delays taking place at the IP level and lower). Network utilization is calculated by the size of the message and the bandwidth of the network *NetBW*. *MsgLossRate* is the percentage of physical messages that are lost. This means each physical message encounters a delay of *SendMsgTime* + *BasicNetworkDelay* + Network Utilization Time + *RcvMsgTime*. Note that the message delay experienced by the application will be longer since a multicast message might consist – depending on the order and reliability – of several physical messages and additional delays occur (e.g., messages wait in buffers or the site must wait for the token).

5.1.3 Transactions

We distinguish between different transaction types where each type is determined by a number of parameters. Each transaction performs *TransSize* operations. We distinguish between read and write operations. *WriteAccessPerc* is the percentage of write operations of a transaction. *RWDepPerc* (ReadWriteDependency) determines the percentage of read operations on objects that will be written later. These require read locks that are kept until the end of the transaction even when using cursor stability. If *WriteAccessPerc* is zero, the transaction type describes queries (read-only transactions) that can be performed locally. *TransTypePerc* gives the percentage of the workload that belongs to each transaction type.

Transaction execution and concurrency control are modeled according to the algorithms described in Section 4.1. When a transaction is initiated, it is assigned to one node. All read operations are performed sequentially at that node. Write operations are delayed or performed on shadow copies. At the end of the transaction, the write set is sent to all nodes and executed at all sites. For comparison purposes, we have also implemented a traditional distributed locking protocol using ROWA with strict 2-phase-locking (2PL). In this case, each read operation is executed locally and each write operation is multicast to all sites using the basic reliable multicast. At the remote sites, whenever the lock for the operation is acquired, an acknowledgment is sent back (note that this is an optimization to the standard protocol where the acknowledgment is not sent before the entire operation is executed). When the local site has received all acknowledgments and executed the operation the next operation can start. Whenever a deadlock is detected at a site, a negative acknowledgment is sent back to the local node which, in turn, will multicast an abort message to the remote nodes. When all operations are successfully executed, the local site sends a commit message to the remote sites.

To deal with the deadlock behavior of 2PL, we have implemented two versions of distributed deadlock detection. As a first possibility, distributed deadlocks are detected via timeout as it is usually done in commercial systems. The parameter *Timeout* sets the timeout interval. As a second possibility, we have implemented a global deadlock detection mechanism. Whenever a node requests a lock and the lock must wait, the detection algorithm is run. In our simulation, this algorithm is "ideal" in the sense that it detects deadlocks instantaneously. That is, it does neither consume any resources (CPU or disk) nor exchange messages.

A transaction that is aborted is restarted immediately and makes the same data accesses as its original incarnation. We use an open queuing model. At each node, transactions are started according to an exponential arrival distribution with a mean determined by *InterArrival*. The *InterArrival* parameter determines the throughput (transactions per second) in the system (e.g., small arrival times lead to high throughput). Hence, the number of local transactions active at a node (multiprogramming level) varies, depending on how long transactions need to execute. It is possible but not required to set a *Mpl* parameter to limit the multiprogramming level. This means that whenever the multiprogramming level exceeds the *Mpl* threshold, new transactions are not started but enqueued.

DBSize	10000
NumSites	10 (except exp. 4)
NumDisks	10
ObjectAccessTime	0.2 ms
DiskAccessTime	20 ms
BufferHitRatio	80%
NetBW	100 Mb/s
MsgLossRate	2%

Table 5.2: Baseline parameter settings

5.2 Experimental Setting

Some of the parameters are fixed for most of the experiments discussed since their variation led only to changes in absolute but not in relative behavior of the protocols. Their baseline settings are shown in Tables 5.2 and 5.3.

The database consists of 10,000 objects. The number of nodes is fixed to 10 except for the last experiment. The number of disks is 10 per site. CPU time per operation is 0.2 ms (= milliseconds) and disk access takes 20 ms. The buffer hit ratio is fixed to 80% and the network has a bandwidth of 100 Mb/s. Note that bandwidth was never a limiting factor in our experiments. This matches results from previous studies [FvR95a, MMSA^{+}96]. Furthermore, we assume a 2% message loss rate for all of our experiments.

We use three different transaction types. Two of these are update transactions: *short* transactions, consisting of 10 operations, and *long* transactions, consisting of 30 operations. Both types have an update rate of 40% and a read/write dependency of 30%. Short transactions represent a workload with low data contention (conflict rate), whereas long transactions show higher data contention. The third transaction type is a query (read-only transaction) with 30 operations. The timeout interval for the distributed 2PL algorithm is 1000 ms for short update transactions. For long update transactions and queries the global deadlock detection mechanism is used. The first experiment provides a throughput analysis, i.e., the inter-arrival time is continuously decreased thereby increasing the workload until the system saturates. In all other experiments the throughput is fixed (by choosing a specific inter-arrival time according to the transaction type) in order to evaluate the influence of other parameters. This is done such that the pure overhead of executing operations (CPU/disk) at each single site is about the same for all experiments and within reasonable boundaries (we did not want the transaction processing CPU/disk to be the bottleneck resource). For instance, since queries are only executed locally while the write operations of update transactions are executed everywhere, a system can achieve a higher throughput with queries. Hence, we set the inter-arrival times of a mixed query/update workload (80 ms) smaller than for a workload with only long update transactions (120 ms).

The main performance metric is the average response time, i.e., the average time a transaction takes from its start until completion. These average response times are with 95% confidence within a 5% interval of the shown results. The response time of a transaction consists of the execution time (CPU + I/O), waiting time and the communication delay. In addition, abort rates of each protocol are also evaluated in order to provide a more complete picture of the results. In the performance figures and the discussions, the following abbreviations are used: SER for serializability, CS for cursor stability, SI for snapshot isolation and HYB for the hybrid protocol. D is a shortcut for the deferred writes approach, SH for the shadow copy approach.

Transaction Type	Short	Long	Query
TransSize	10	30	30
WriteAccessPerc	40%	40%	0%
RWDepPerc	30%	30%	0%
Timeout	1000 ms	–	–

Table 5.3: Transaction types

Experiments	Exp. 1	Exp. 2	Exp. 3	Exp. 4
# of Servers	10	10	10	1-100
Txn Type	Long	Short/Long	Long/Query	Short/Query
Throughput	Varying	200 tps/80 tps	125 tps	varying
% of Upd. Txn.	100%	100%	varying	varying
Communication Costs	low	varying	low	low

Table 5.4: Parameters settings of the different experiments

Furthermore, UR indicates the protocols using uniform reliable delivery for all messages (write set WS and decision message c/a). URR indicates the protocols where WS is uniform reliable, c/a are reliable (does not exist for SI) and R refers to the reconciliation based versions where all messages are only reliable.

In all the following figures the order of the labels usually corresponds to the order of the curves with the label of the highest curve always on the top followed by the label of the second highest curve and so on.

5.3 Experiments

This section presents four experiment suites. Table 5.4 provides a summary of the parameter settings for the different experiments. The first experiment provides a general throughput analysis comparing the deferred writes and the shadow copy approach. The second experiment analyzes the impact of the communication overhead, the transaction size and the conflict rate as well as how these three parameters influence each other. The third experiment evaluates the performance of the protocols for a mixed workload consisting of update transactions and queries and analyzes how these two transaction types affect each other. Finally, the fourth experiment analyzes the scalability of the system when the number of sites increases up to 100 nodes. The last three experiments are only conducted with the deferred writes approach. This decision has mainly be due to clarity reasons. The relative performance differences for the varying levels of isolation and fault-tolerance have shown to be the same for both the deferred writes and the shadow copy approach. Hence, we prefer to only depict the results for one approach in order to focus on the relevant issues. While we only shortly discuss the behavior of 2PL along with the experiments, we provide a longer discussion in a special subsection in order to point out the reasons why standard 2PL behaves so different than our protocols.

5.3.1 Experiment 1: Shadow Copies vs. Deferred Writes

This experiment provides a first throughput analysis in order to give a feeling of the effect of the different levels of isolation SER, SI and CS. Furthermore, it compares the deferred writes (D) and the shadow copy approach (SH). It also provides a comparison with standard distributed 2-phase locking (2PL). The levels of fault-tolerance will be compared thoroughly in the next experiments. In here, we only use reliable message delivery.

The configuration is chosen to represent a LAN cluster and consists of 10 nodes. The communication parameters *BasicNetworkDelay, SendMsgTime* and *RcvMsgTime* are set to values such that they result in effective message delays that are equivalent to those measured in a real cluster network. For further details, see experiment 2, in particular, test run IV of Table 5.5. The workload represents a very high conflict OLTP workload consisting only of long update transactions. With such a workload we are able to better depict the differences between the different levels of isolation.

For the shadow copy approach we simulate two types of update propagation. In the first type (denoted as *write* in the figures), write operations are performed at all sites (local and remote) and hence, have the same CPU and disk overhead at all sites (0.2 ms for CPU, 20 ms for disk access). In the second type (denoted *apply* in the figures), the local site sends the physical updates and remote sites only apply the updates. We assume that applying updates at a remote site has only half the CPU and disk costs than performing the operation. For the deferred writes approach, it is not possible to send the physical updates and hence, write operations have always the full CPU and disk costs at all sites. We would like to note that our model to reduce resource consumption in the case of applying the physical updates is rather vague and simplified. How much processing capacity we can really safe depends strongly on the system and the type of operation. We will evaluate this issue in much more detail in Chapter 6. In here, we only want to give a rough intuition of how resource consumption effects performance.

Figure 5.1 depicts the response times and Figure 5.2 the abort rates for long transactions at increasing throughput. For all protocols, response times and abort rates increase with the throughput until the maximum throughput is reached and the response time degrades. For some of the protocols the maximum throughput is determined by disk saturation. These are the protocols with generally low conflict rates, i.e., CS-SH-apply and CS-D. These protocols are able to take full advantage of the hardware resources without limiting concurrency. For other protocols degradation is due to a combination of nearly saturated resources and aborts. These are, the CS-SH-write, SER-D, SER-SH-apply, SI-D and SI-SH-apply. For SER-SH-write and SI-SH-write, degradation is mainly due to abort rates with some effect due to resource saturation.

Comparing the response times of the different approaches to execute write operations, the shadow approach applying the updates (SH-apply) is better than the deferred writes approach (D) for all levels of isolation, and D is generally better than the shadow copy approach executing the operations (SH-write). For D and SH-write, 120 tps are the maximum throughput achievable at which resources are nearly completely saturated. Further increase leads to response time degradation. D and SH-write have the same resource consumption and the worse behavior of SH-write is mainly due to the higher abort rates: SH-write requests RIW locks that are vulnerable for abort and has longer read phases than D. This is, however, not the case for SH-apply. The good performance of SH-apply is mainly due to the reduced resource consumption since remote sites only apply the updates. In our setting, for instance, disk utilization is reduced by a third. This keeps response times small and helps to reduce the abort rates to a certain degree. However, SER and SI are not able to achieve higher throughput rates than the presented 120 tps due to the higher abort rates. Only CS-SH-apply is able to support higher throughputs (in our configuration up to 180 tps) before it is limited by a combination of high abort rates and disk saturation.

Looking at each of the approaches D, SH-apply, and SH-write individually, the performance of the different isolation levels within each approach is directly correlated with the abort rate. For instance, for the deferred writes approach, CS behaves better than SI and SER, SI and SER behave similar for low and moderate

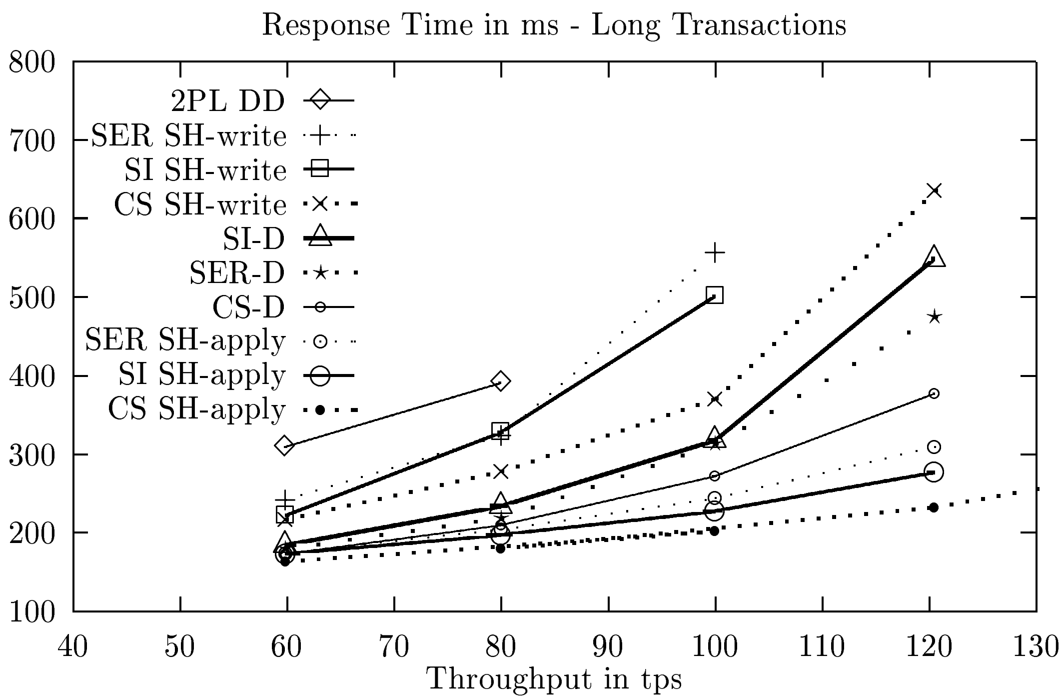

Figure 5.1: Experiment 1: *Response time* of long transactions at increasing throughput

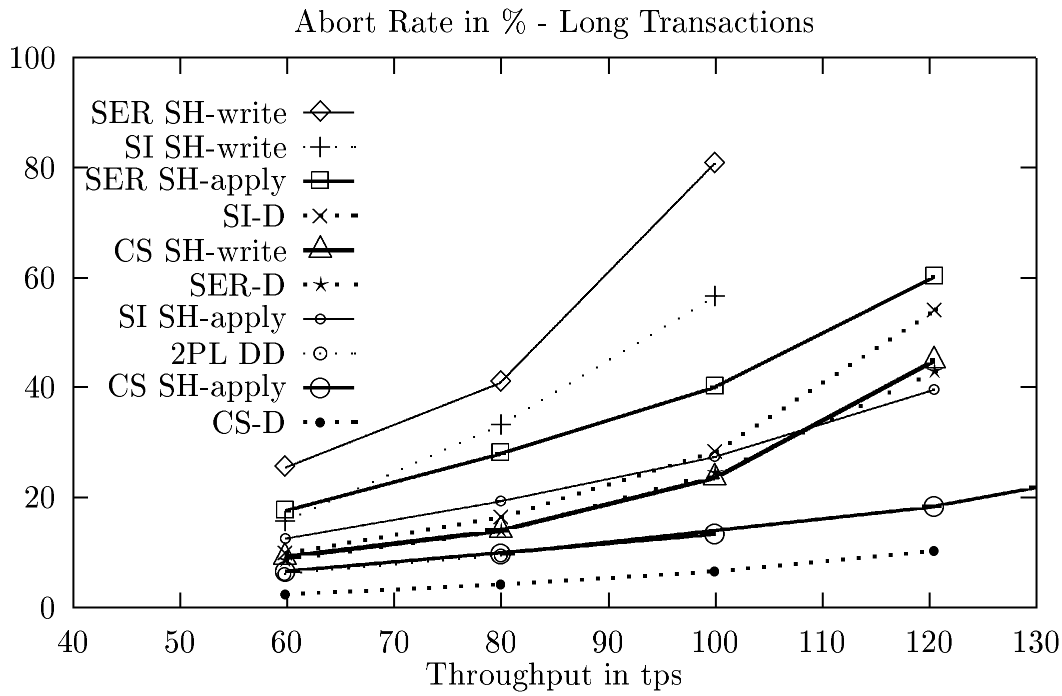

Figure 5.2: Experiment 1: *Abort rate* of long transactions at increasing throughput

throughput, and SER is better than SI for high throughput. In D, CS aborts only very seldomly long-read/write conflicts, SI aborts upon write/write, and SER aborts upon read/write. In the specific distribution of read and writes in this workload, write/write conflicts occur more often than read/write and SI has more aborts than SER. Note that in all cases, the abort rates are rather high. This is due to the chosen workload of long transactions.

Looking at each level of isolation individually, we can observe different abort behavior. Within SER and CS, both shadow approaches have higher abort rates than deferred writes. This means, the RIW locks and not the response time have the biggest impact on the conflict rate. This is different for SI. At low throughput, the shadow copy approaches have also higher abort rates than deferred writes, but at high throughput, applying the writes has smaller abort rates than deferring them. Here, the only determining factor for the abort rate is the length of the read phase. The smaller the read phase, the smaller is the probability that there are concurrent transactions writing the same items.

2PL has by far the worst response time and degrades very fast. It is only able to support very low throughput rates. Although the abort rates are generally low, the long blocking times when transactions have to wait for locks increase response times very fast. All the protocols proposed in this paper avoid this problem – CS and SI mostly avoid read/write conflicts and SER aborts and restarts readers in a very early phase of transaction execution. Furthermore, choosing the right timeout interval for deadlocks is very difficult and we choose to implement a no-cost deadlock detection mechanism (which is unrealistic in a real system) in order to figure out the "true" abort rate.

Analysis This experiment has provided a first performance analysis of the protocols. To summarize, all the protocols proposed in this thesis have significantly smaller response times and higher maximum throughputs than the traditional 2PL protocol proving that our approach applies the adequate mechanisms needed for efficient eager replication and is able to support high update rates with acceptable performance.

All proposed protocols behave similarly when throughput is low. At higher throughputs the results show the behavior as predicted in Chapter 4. In regard to the isolation levels, CS has significantly lower conflict rates than SER and SI. SER and SI have very similar conflict rates and the differences strongly depend on whether there are more read/write or more write/write conflicts in the workload. Using shadow copies results in higher abort rates than deferring writes, applying physical updates reduces resource consumption and with it, response times. The exact performance gain of applying physical updates, however, depends strongly on the concrete implementation, and can not be modeled adequately in a simulation system. Nevertheless, we believe that applying the physical updates will be the preferable choice in most configurations. The other alternative to handle high update rates is using cursor stability in order to alleviate the conflict problem.

This experiment has only shown the performance results for long transactions with high conflict rates. Tests with short transactions have shown the same relative behavior of the protocols. However, the differences between the different isolation levels were much smaller and nearly all protocols were able to handle throughput rates up to the resource saturation point without degradation of the abort rates.

The next experiments are all based on the deferred writes approach. Since the relative performance of the different isolation levels has shown to be similar for both the deferred writes and the shadow copy approach, we believe it is sufficient to only show further results for one of them. Only by restricting us to one approach we are able to depict further interesting performance aspects without overloading the figures. We chose to use the deferred writes approach, since the performance of the shadow copy approach strongly depends on how we model the costs of applying physical updates, a topic we prefer to not investigate any further in our simulation study. Chapters 6 and 7 evaluate this issue in more detail.

Test Run	I	II	III	IV	V	VI	VII	VIII	IX
BasicNetworkDelay	0.01	0.05	0.1	0.2	0.4	0.5	0.6	0.8	1.0
SendMsgTime									
Small Msg.	0.005	0.025	0.05	0.1	0.2	0.25	0.3	0.4	0.5
Medium Msg.	0.01	0.05	0.1	0.2	0.4	0.5	0.6	0.8	1.0
Large Msg.	0.02	0.1	0.2	0.4	0.8	1.0	1.2	1.6	2.0
RecvMsgTime									
Small Msg.	0.01	0.05	0.1	0.2	0.4	0.5	0.6	0.8	1.0
Medium Msg.	0.02	0.1	0.2	0.4	0.8	1.0	1.2	1.6	2.0
Large Msg.	0.04	0.2	0.4	0.8	1.6	2.0	2.4	3.2	4.0

Table 5.5: Communication settings in ms

5.3.2 Experiment 2: Communication Overhead vs. Concurrency Control

The proposed protocols differ in the number of messages and their delays. Whereas the SI protocol uses a single totally ordered multicast message per transaction, the SER/CS/Hybrid protocols send one message using the total order service and one message using the basic order multicast per transaction. The protocols using uniform reliable message delivery suffer from a higher message delay than the protocols using reliable message delay. In addition, the protocols use different concurrency control mechanisms providing different conflict profiles.

In the second experiment we analyze the interplay between these aspects. To do so, we vary the communication parameters to model both efficient communication with small message delays and little overhead (typical for LANs) and slow communication with long delays and high overhead (as in WANs). Using this, we want to analyze the sensitivity of the protocols to the communication overhead. By looking at two different workloads (short and long transactions) we are able to judge which optimization is more effective: reducing message overhead or reducing conflict rates.

Communication is determined by the parameters *BasicNetworkDelay*, *SendMsgTime* and *RcvMsgTime.* Table 5.5 depicts their settings in ms for this experiment varying them from little to high overhead. We have set different overhead values for small messages (acknowledgments, commit and abort messages), medium size messages (with one write operation in the case of distributed 2PL) and large messages (the entire write set). Furthermore, we have set the overhead of point-to-point messages to half of the overhead of a multicast message assuming that there is less flow control involved with point-to-point messages. Note that the delay of a multicast message will be much longer than the individual values depicted in the test runs, since the minimum delay is calculated by the sum of these values plus additional waiting times. For instance, using the settings of test run IV, the delay for a basic multicast is 1 ms, for a total multicast 7 ms. These are values that are equivalent to measurements on real networks using between 6 and 10 nodes. In the performance figures of this experiment we depict the different test runs by using the corresponding setting of the parameter *BasicNetworkDelay*.

Short Transactions Figure 5.3 shows the response times and Figure 5.4 the abort rates for short transactions for an inter-arrival time of 50 ms per node (i.e., around 200 short transactions per second in the system). At such a workload, transaction processing requires few resources and data contention is small. Hence, the message overhead has a great impact on the overall response time (Figure 5.3). With a low

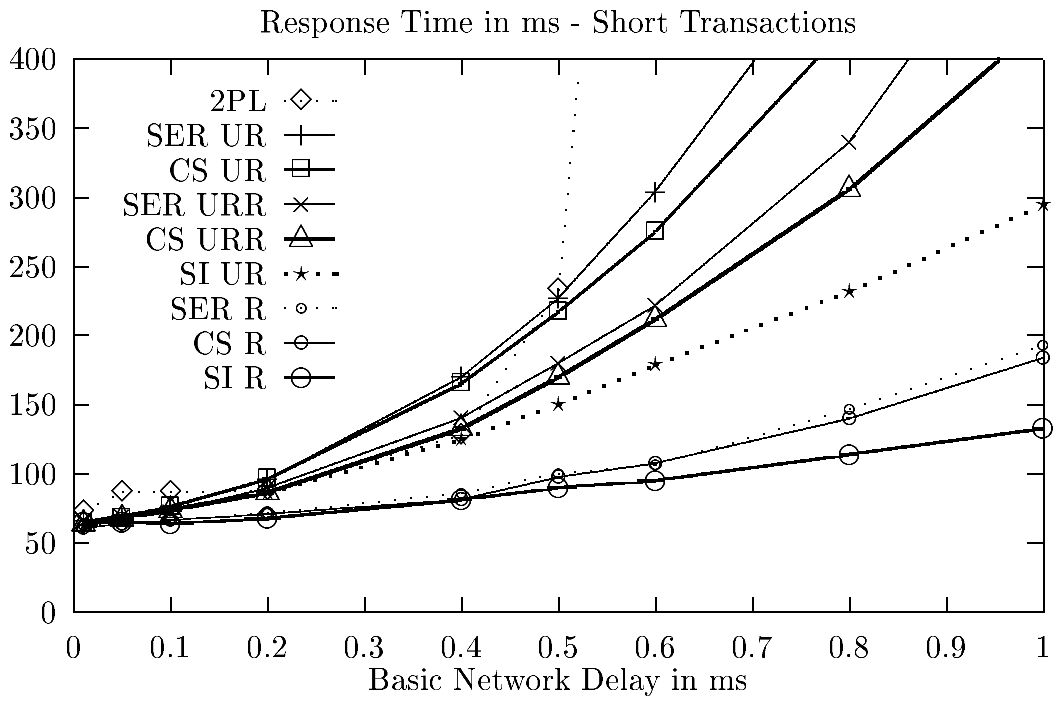

Figure 5.3: Experiment 2: *Response time* of short transactions

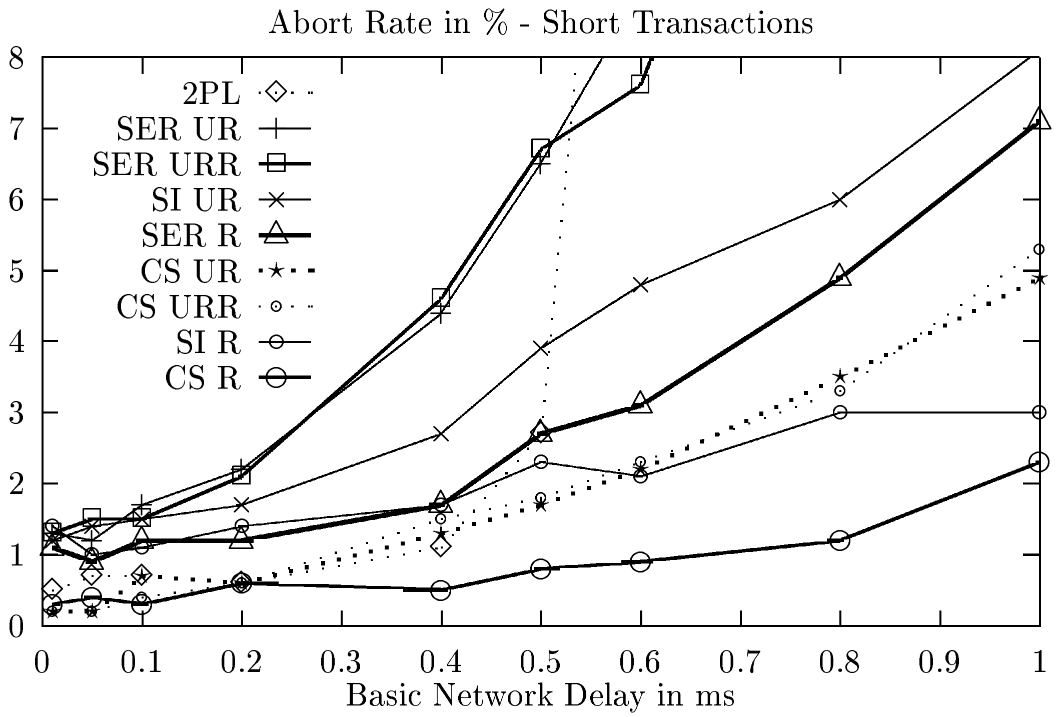

Figure 5.4: Experiment 2: *Abort rate* of short transactions

communication delay, the response time corresponds to the execution time of the transaction. With an increasing communication overhead, the response time of the different protocols increases depending on the number and complexity of the message exchanges. The communication processors are becoming more and more utilized and are nearly saturated at high communication costs. Hence, when communication is costly and the delay long, the R protocols show best performance (SI outperforming SER and CS since it needs only one multicast message). A little worse is SI-UR. It performs even better than SER-URR and CS-URR due to the reduced number of messages. Of the suggested protocols, SER-UR and CS-UR have the worst behavior since they wait to deliver both the write set and the decision message until all nodes have sent acknowledgments.

2PL has a performance similar to the other protocols when communication is fast. However, when communication is more expensive, 2PL degrades very quickly due to the enormous amount of messages. Although we modeled point-to-point messages with lower overhead than broadcast messages and considered that sending only one write operation is cheaper than sending the whole write set, the communication processor saturates.

The abort rates (Figure 5.4) are generally low for all protocols. The fact that abort rates increase as the communication delay increases is explained by the number of transactions in the system. Slow communication delays transactions and causes transactions to spend more time in the system. As more transactions coexist, the probability of conflicts increases, and with it, the abort rate. Therefore, the R protocols have a lower abort rate than the UR and URR versions since they shorten the time from BOT to the delivery of the write set, thereby reducing the conflict profile of the transaction. Generally, since the experiment has a skew towards write operations, CS has the lowest abort rates. SER and SI have similar abort rates with fast communication, but as the communication delay increases the behavior of SER-UR and SER-URR degrades. This is due to the abort of readers upon arrival of a write transaction which is later also aborted. Aborting the readers was unnecessary, but, as the communication delay increases, the likelihood of such cases increases. SI does not have this problem since the decision on abort or commit can be done independently at each node and a transaction only acquires locks when it is able to commit. Note also, that the UR and URR versions of SER (and also the UR and URR versions of CS) behave similarly. The reason is that, in these protocols, a transaction can only be aborted when it is in its read phase or when it waits for its write set to be delivered. These phases are the same in both the UR and URR versions of the protocols.

With low communication costs, 2PL has lower abort rates than SER and SI, since SER and SI sometimes abort readers/writers unnecessarily. However, the abort rates of 2PL quickly degrade when response times become too long due to the saturation of the communication processor.

Long Transactions Figure 5.5 shows the response times and Figure 5.6 the abort rates for long transactions for inter-arrival times of 120 ms (i.e., around 80 long transactions per second in the system). Long transactions have higher data contention than short transactions. However, the total number of messages is smaller since less transactions start per time interval. Although the reliability of message delivery is still the dominant factor for high communication costs, the concurrency control method becomes a more important factor in terms of response time (Figure 5.5). Looking at the R protocols, CS outperforms SI and SER due to its low conflict rate. The advantage of SI sending only one message is not the predominant factor, since communication is less saturated and the message delay itself has not a large direct impact on the response time of long transactions. Furthermore, the URR and UR versions of CS are better than SI-UR, SER-UR and SER-URR. The last two protocols do not perform well for slow communication due to the high conflict rate.

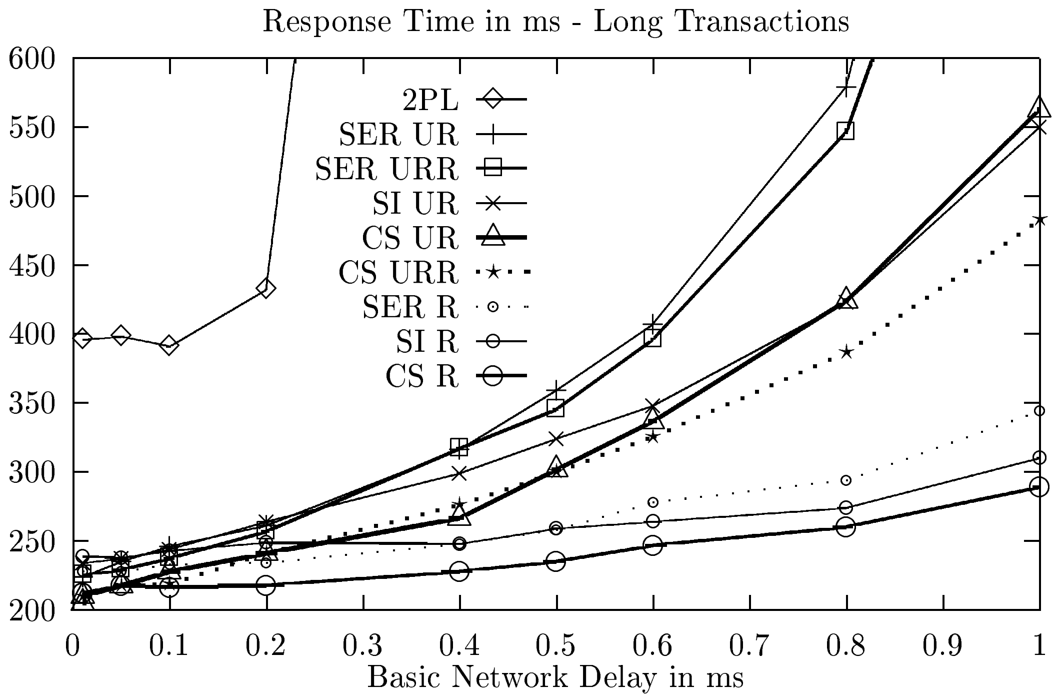

Figure 5.5: Experiment 2: *Response time* of long transactions

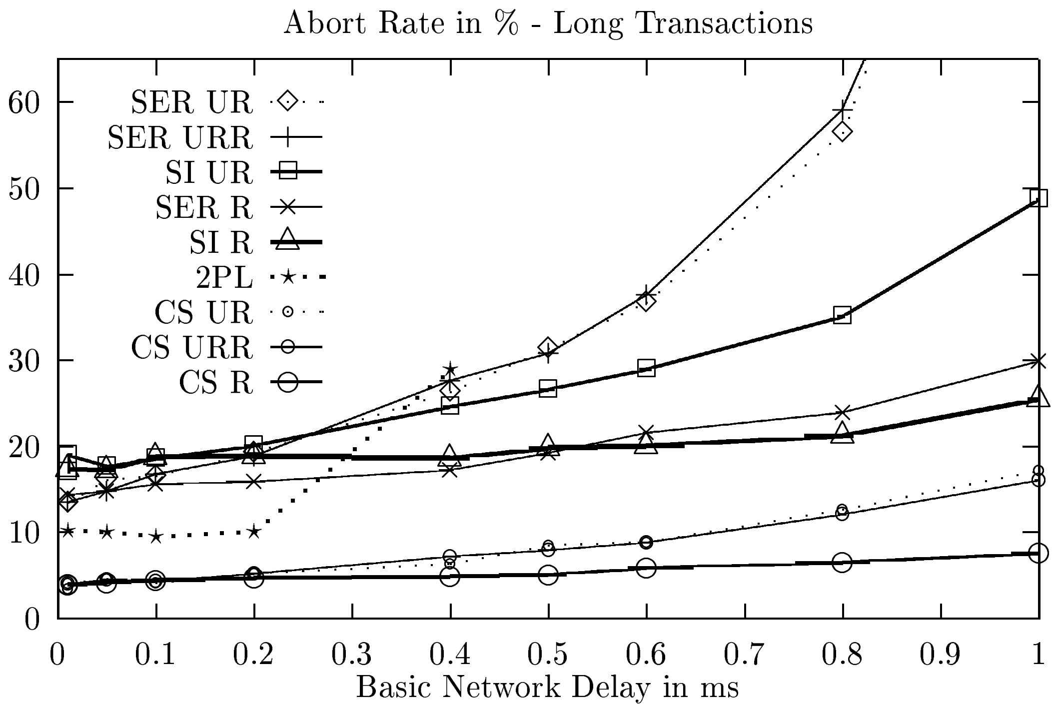

Figure 5.6: Experiment 2: *Abort rate* of long transactions

Looking at the abort rates (Figure 5.6), CS clearly outperforms all other protocols due to its low conflict rate. Furthermore, there is no degradation when communication becomes slow. SER-UR, SER-URR and SI-UR, however, degrade when communication takes longer and here even the R versions of SER and SI have rather high abort rates. However, these abort rates do not have a large impact on the response time since they usually happen in an early phase of the transaction. Nevertheless, they might cause a problem if the system cannot restart aborted transactions automatically but instead only returns a "transaction failed" notification to the user. If this is the case, CS might be the preferable choice.

The response time of 2PL degrades very fast. Even with fast communication, it behaves significantly worse than the other protocols. The delay created by acquiring locks on all sites increases the conflict rate extremely and prohibits short response times.

Analysis For the proposed protocols the general behavior can be summarized as follows. In "ideal" environments, the behavior of the protocols is very much the same in all cases and serializability and uniform reliable message delivery can be used to provide full consistency and correctness. However, as soon as conflict rates or network delay increase, both serializability and uniform reliable message delivery might be bad choices: they result in increasing abort rates and longer response times. The results show which strategy to follow depending on the characteristics of the system. Thus, if the communication system is slow, performance can only be maintained by choosing protocols with low message overhead like snapshot isolation or reliable message delivery. Similarly, if data contention is high, lower levels of isolation are the only alternative to keep abort rates small.

Generally, 2PL shows worse performance than any of the proposed algorithms. It is very sensitive to the capacity and performance of the communication system, it is not able to handle high conflict rates and it degrades very fast when conditions worsen.

5.3.3 Experiment 3: Queries

In practice, replication pays off when the majority of the transactions are queries which can be executed locally without any communication costs. We have analyzed the behavior of the protocols using a mixed workload consisting of long update transactions and queries. The percentage *TransTypePerc* of both transactions types are varied between 10% to 90%. Since we want to investigate pure resource and data contention and not the impact of the communication overhead, the communication costs are set to be low (see values of test run III in Table 5.5) and we only show the results for the R-protocols. We also analyze the hybrid protocol using SER for updating transactions and a snapshot for queries.

Figures 5.7 and 5.8 show response times and abort rates for update transactions. Figures 5.9 and 5.10 show response times and abort rates for queries. The results are shown as a function of the percentage of queries in the workload for an inter-arrival time of 80 ms (i.e., around 125 transactions per second in the system). The response times for both transaction types decrease when the percentage of queries increases due to less resource contention (less replicated write operations, more local read operations) and less data contention (shorter lock waiting times, lower abort rates). The differences in the proposed protocols directly reflect the different abort rates for writers and readers of the different protocols. For update transactions (Figures 5.7 and 5.8) CS behaves better than the others for low query rates since data contention is rather high and CS has less aborts than the others. SER, HYB and SI behave similarly having very similar abort rates. 2PL does not admit low query rates and degrades for query rates smaller than 50%. The response times of update transactions are longer than with the other protocols due to the additional message exchange and longer waiting times. Abort rates, however, are smaller, since only deadlocked transactions are aborted.

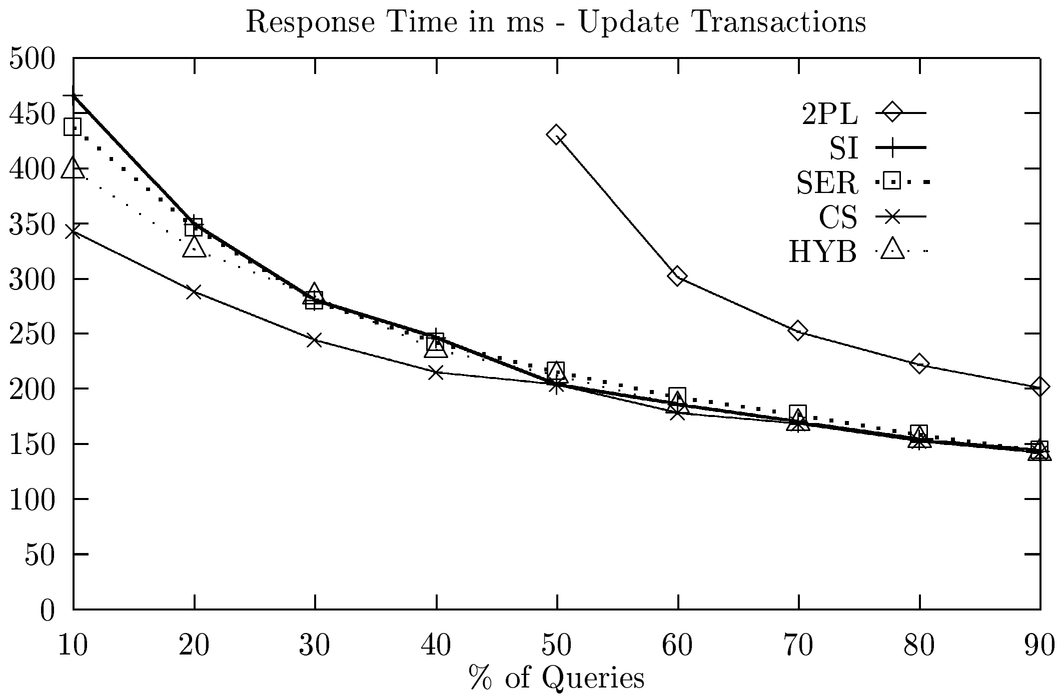

Figure 5.7: Experiment 3: *Response time* for update transactions

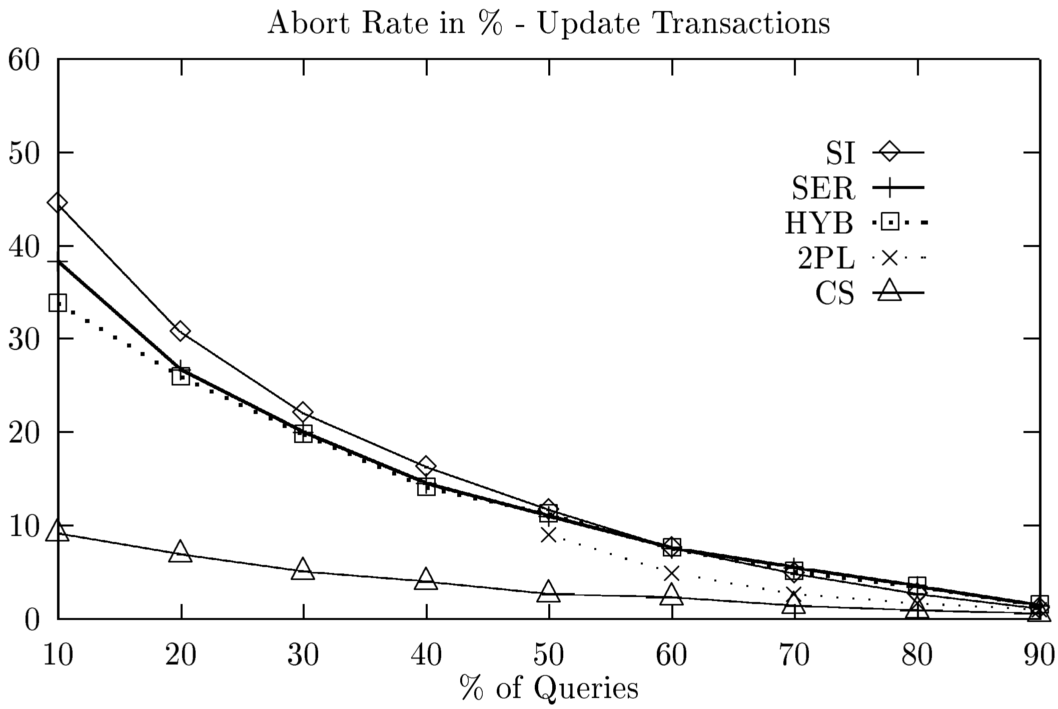

Figure 5.8: Experiment 3: *Abort rate* for update transactions

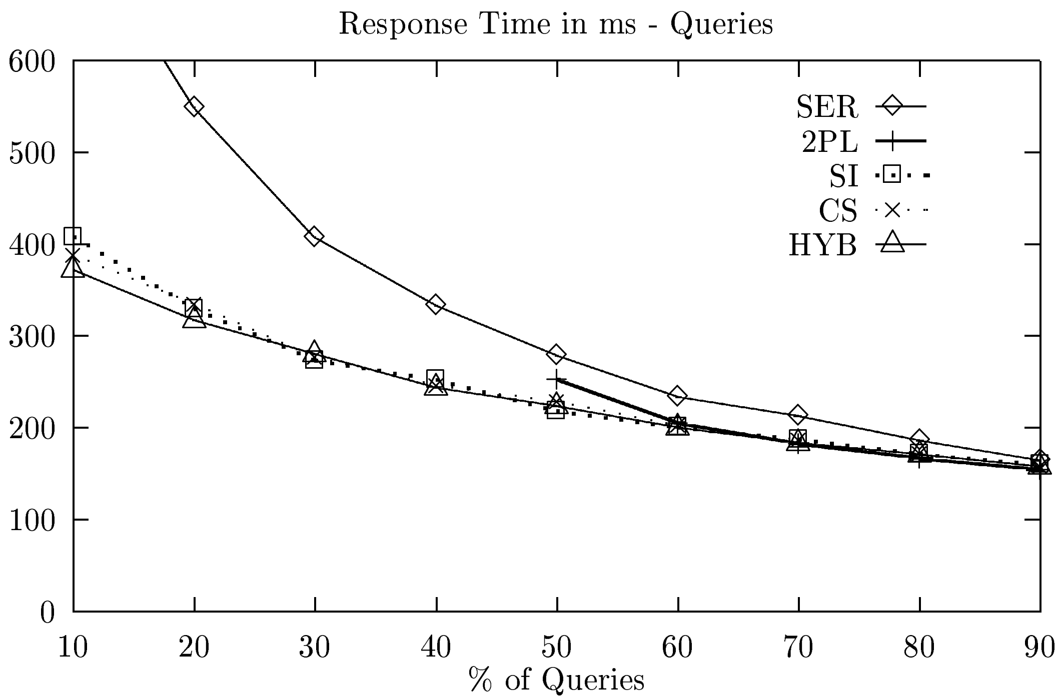

Figure 5.9: Experiment 3: *Response time* for queries

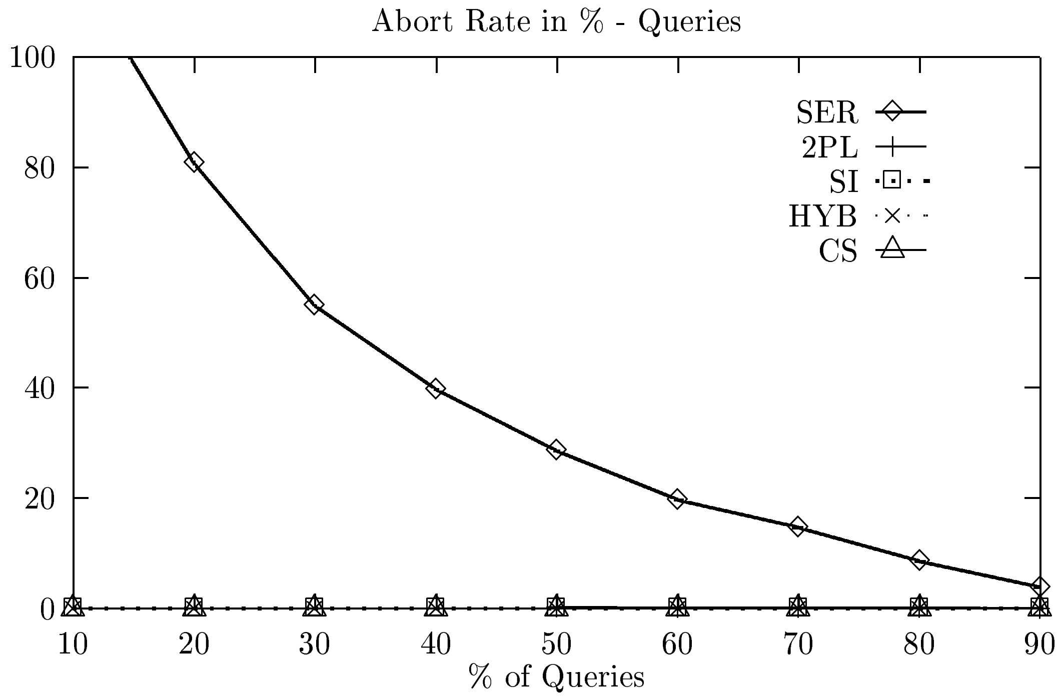

Figure 5.10: Experiment 3: *Abort rate* for queries

For queries (Figures 5.9 and 5.10), SER has worse response times than the other protocols. SER is the only protocol that aborts readers (even 2PL has very low abort rates for queries). If many update transactions are in the system, SER has very high abort rates and hence high response times. However, the abort rate, and with it the response times, decrease very fast with an increasing number of queries. Here, response times start to be comparable with the results of the other protocols. 2PL, as long as it does not degrade, can provide the same performance as the other protocols. However, the response time of the queries strongly depends on the performance of the update transactions.

Analysis This experiment clearly shows that queries need a special treatment to avoid unnecessary aborts. The rather simple SER approach, where potential deadlocks are resolved by aborting transactions, results in an unacceptable high abort rate for queries (40% in the case of 40% queries). Therefore, the hybrid protocol using a snapshot for readers seems a good alternative. Since readers are never aborted, queries yield good performance and do not require excessive resources. For updating transactions, the hybrid protocol provides serializability, unlike SI or CS. However, transactions must be declared read-only in advance to allow this special treatment. Although CS has the best performance results, it does not provide repeatable reads; this may be problematic in certain applications.

This experiment again shows that the applicability of 2PL is restricted to low conflict, low workload configurations and that it degrades very fast if the conditions are not optimal.

5.3.4 Experiment 4: Scalability

The ability to scale up the system depends on the number of update operations in the system since they are executed on all sites. To analyze this factor we run an experiment with a workload of 20% short update transactions and 80% queries with an inter-arrival time of 40 ms per node. As calculated in Section 2.5.1, the scalability is limited with such a workload. Since all update operations are executed on all sites, increasing the number of nodes in the system results in an increasing number of write operations. This finally leads to resource (CPU, disk) saturation. In our configuration using the proposed algorithms, this resource saturation starts at 60 nodes. Further increase in the number of nodes leads to performance degradation. This degradation is only due to the enormous amount of transaction processing power needed to perform the write operations at all nodes. 2PL, on the other hand, scales up only to 20 due to increasing conflict rates.

In a second experiment we chose a decreasing update rate in order to analyze other factors affecting scalability. For example, the number of sites plays an important role since it influences the number of messages involved and, above all, the calculations involved in determining the total order. Again, the workload is a combination of short update transactions and queries. The inter-arrival time of update transactions is kept constant (4 ms) for the entire system, i.e., at a 10-node-system the inter-arrival time is 40 ms per node, whereas in a 100-node-system the rate is 400 ms per node (representing 250 transactions per second in the system). The inter-arrival time of the queries is always 100 ms per node (i.e., 10 queries per second per node). The workload represents an application where a formerly centralized OLTP database is used in a distributed manner (that means the same amount of write accesses is distributed over more nodes) and the analytical access (read operations) increases with the number of nodes. The communication parameters were set to the values of test run IV in Table 5.5. We skipped the SER protocols since we saw already in the previous experiment that SER aborts queries too often.

Figures 5.11 and 5.12 depict the response times for update transactions and queries as the number of sites in the system increases from 5 to 100. For update transactions (Figure 5.11) the 5-node system behaves

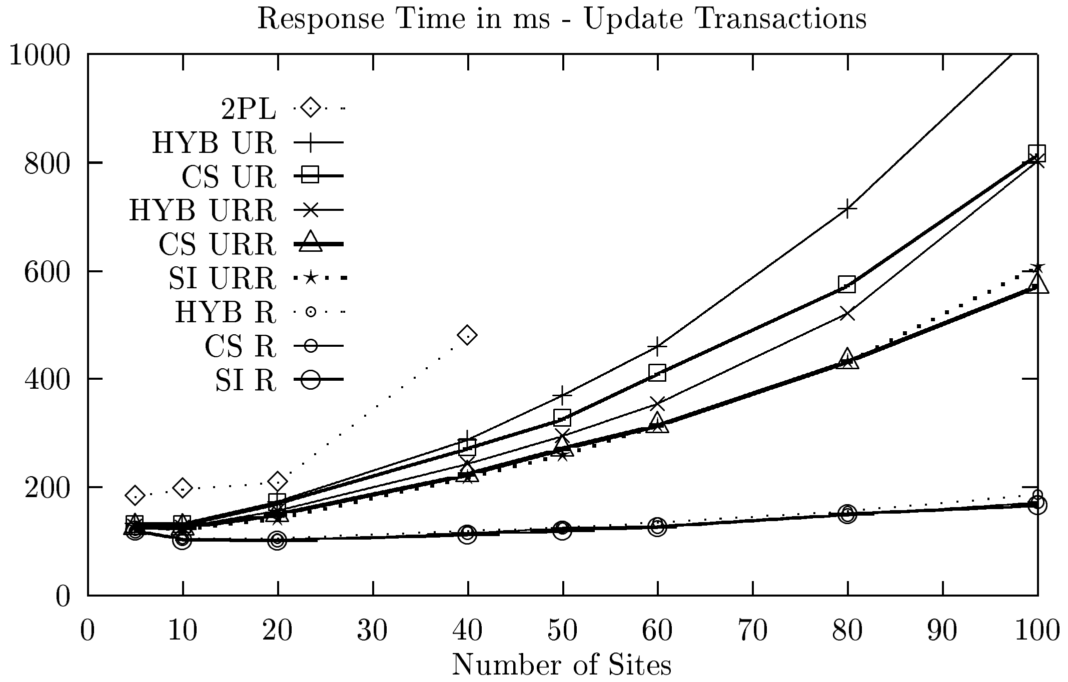

Figure 5.11: Experiment 4: Response time of *update transactions* for different number of nodes

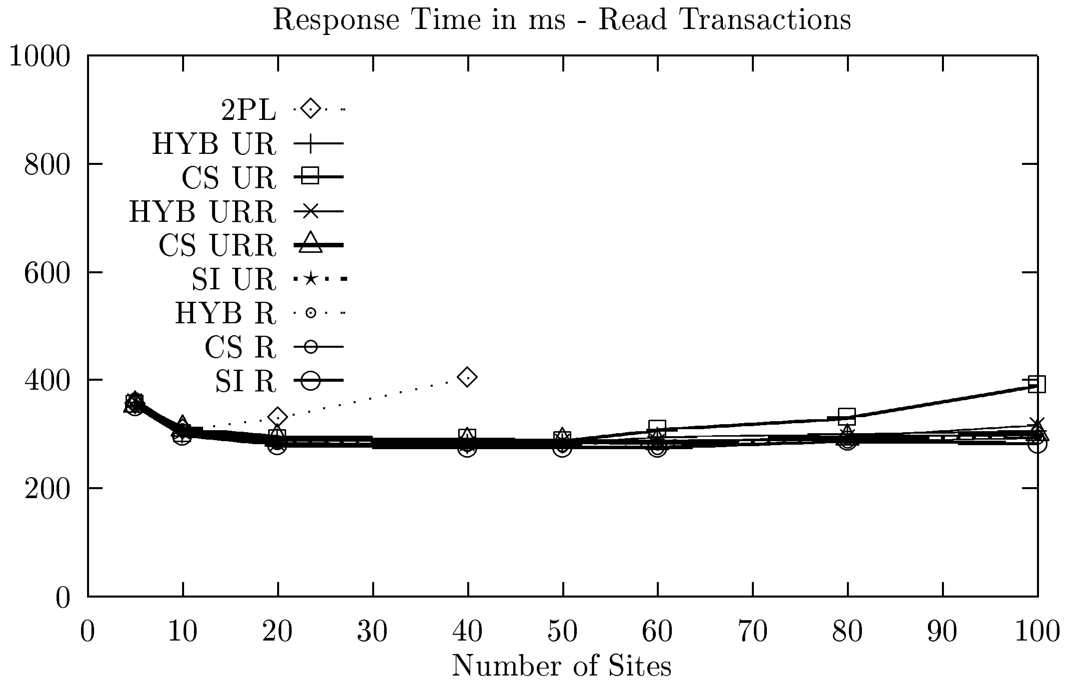

Figure 5.12: Experiment 4: Response time of *queries* for different number of nodes

slightly worse than the 10-node system due to the higher load (each node executes the read operations of 50 update transactions per second while in the 10-node system each node executes the reads of 25 update transactions per second). For ten nodes and higher we can observe a very similar behavior as observed when increasing communication overhead (see Figure 5.3). The response time for all protocols increases with the number of nodes due to the increased message delay (determining the total order and uniform reliable message delay take significantly more time when the number of nodes increase). The R protocols behave better than their fault-tolerant counterparts and if complex communication protocols are used (URR and UR), concurrency control methods with less conflict rate (CS) or less messages (SI) show better results. Compared to the results in Figure 5.3, only SI behaves a little bit worse, i.e., sending only one message does not have an impact since the communication processor is not overloaded. 2PL has worse behavior than the other protocols already for a small number of nodes. The problem is that each write operation has to wait for all the nodes to respond and with each additional node the probability increases that a write operation has to wait on one of the sites for a query to release its locks.

The response time for queries (Figure 5.12) is similar for all developed protocols. Since none of the protocols considered in this experiment aborts queries, they are barely influenced by the behavior of update transactions. Only CS-UR gets slightly worse when the number of sites increases since CS acquires read locks for queries and has to wait for update transactions to release their write locks. When communication takes longer the UR version keeps write locks longer than the URR and R versions. Write locks are kept until all write operations have been performed and the commit message has been delivered. Since URR and R send the commit message only with reliable delivery the message is delivered as soon at it is physically received. With UR however, the commit message is only delivered when all acknowledgments have been received. Hence, when the number of sites in the system increases it takes longer to receive all acknowledgments and therefore, UR blocks waiting read locks longer than URR or R. Note that neither the HYB nor the SI protocol acquire read locks and hence have the same results for queries for both UR and R versions. For 2PL, even for 20 nodes the response time of queries is worse compared to the other protocols since queries have to wait long for update transactions to release their locks. From 40 nodes on, the response time increases due to the degrading response times of update transactions.

Analysis The proposed algorithms scale up fairly well, providing good performance for even large system sizes. The main factor to consider is the communication delay (which, of course, increases with the number of sites in the system). Thus, our protocols scale well as long as the communication system scales well, i.e., it provides short message delays although the number of nodes increases.

5.3.5 Behavior of standard 2PL

In this section we would like to shortly discuss the behavior of 2PL especially when conflict rates were high. One problem has been the choice of the timeout interval for deadlocked transactions. When conflict rates are small the timeout interval is not important because the chances are small that a transaction wants to acquire a lock that is hold by a deadlocked transaction. However, when conflict rates are higher it is very difficult to predict the timeout interval. When a transaction has to wait for one lock its response time can easily be doubled. Hence the timeout interval has to be chosen big enough to not abort transactions that simply wait for a lock. However, choosing it too long delays waiting transactions increasing the conflict problem. Since we experienced that it is very hard to find the "optimal" timeout interval and we wanted to figure out the "true" deadlock rate, we implemented the global no-cost deadlock detection mechanism. The results shown in all figures (besides short transactions in the second experiment) are from using this deadlock detection mechanism.

A second problem has been the number of concurrent transactions in the system. As noted in [GHOS96], the increase in length of the transaction due to the execution of every write operation on several sites makes the conflict rate increase very fast. In the worst case, a transaction T_i acquires a read lock on its first operation and a transaction T_j, having already acquired 20 locks wants a write lock on the same object and has to wait. However, T_i will not release the lock for a very long time delaying T_j for the same time and blocking all 20 objects T_j has locks for. More and more transactions enter the system, trying to acquire locks that are hold by these transactions. If this happens we observe a degradation of the system. We encountered this degradation in some cases even for short transactions in experiment 2. While the system runs fine for a long time having, e.g., never more than 5 or 6 local transactions in parallel on each site, all of a sudden an unlucky combination of concurrent transactions delays too many transactions in the system, and within a short time period the number of parallel transactions jumps up to 20 transactions, the deadlock rate explodes and the system degrades. We limited the problem by only allowing a maximum number of parallel transactions in the system (*Mpl* parameter). When the *Mpl* threshold is reached the start of newly submitted transactions will be delayed until one of the active transactions has committed. While this works fine to survive short time periods of increased conflict rate it does not solve the problem in the general case. The reason is that *Mpl* has to be chosen big enough to allow for the necessary throughput, i.e., the number of submitted transactions must on average be the same as the number of committed transactions per time interval. In Figure 5.5, for instance, 2PL achieves the required throughput of around 80 transactions per second up to a *BasicNetworkDelay* of 0.2 ms by allowing up to 3 local transactions running in parallel. For a *BasicNetworkDelay* of 0.4 ms we were not able to achieve a throughput higher than 50 transactions per second for any *Mpl*. Setting MPL to 3 the response time is 620 ms with an abort rate of 12%. Doubling the MPL to 6, the response time increases to 1500 ms and the abort rate to 29%.

5.4 Discussion

This chapter has provided a quantitative performance analysis of the proposed algorithms. The results show that eager update everywhere replication is feasible for a wide spectrum of applications. A comparison with a standard distributed 2PL protocol shows that the proposed protocols have generally shorter response times, allow for higher throughput, are much more stable and perform well for a much broader range of workloads and configurations. They all avoid the fast and abrupt degradation experienced with 2PL/ROWA. Our protocols show good results even when the number of nodes in the system is high. The good performance is achieved by keeping the number of messages small, by using the semantics of group communication systems, and by avoiding deadlocks and a 2-phase commit protocol.

The performance of the proposed protocols differs depending on the system configuration and the workload. Sometimes all protocols perform equally well, sometimes one protocol type outperforms the others, and at extreme situations some protocols degrade later than others. In each situation, we should use the protocol that offers the highest possible degree of isolation and fault-tolerance while still providing acceptable performance. This means, we can use a fault-tolerant, fully serializable protocol under certain conditions: fast communication, low system load, low conflict rates and the percentage of queries is reasonable high. If the system configuration is not ideal, as it will happen in many cases, the optimizations in terms of lower levels of isolation and fault-tolerance help to maintain reasonable performance. Which one to choose depends on the specific configuration:

- In general, high update rates can become a severe problem in a replicated system due to their extensive resource requirements at all sites. To alleviate the problem, conflict rates can be reduced by using lower levels of isolation, e.g. cursor stability or snapshot isolation. Alternatively, if updates are first performed during the local read phase, write operations need not be reexecuted on remote sites, instead they only need to apply the physical changes. This reduces resource contention significantly and hence, provides faster response times.
- Efficient communication plays a major role in eager replication, and the message delay for each individual message must be low. If the network provides fast message transfer, communication does not have a big impact on any of the proposed protocols and uniform reliable message delivery can be used in order to guarantee data consistency on all sites. If the network is slow using reliable message delivery is an attractive alternative to keep response times small. We believe that in most cases the price to be paid – the possibility of inconsistencies on failed nodes – is acceptable.
- Message delay has also an indirect influence on data contention. Long message delays increase the response times of the transactions. This leads to more concurrent transactions in the system, and hence, to higher data contention. This problem is more severe for long than for short transactions. If long message delay cannot be avoided even when reliable message delivery is used, its effect on data contention and abort rates can be alleviated by using concurrency control protocols with lower isolation levels, e.g., cursor stability.
- The results have shown that queries need a special treatment to avoid high abort rates. The hybrid protocol using SER for updating transactions and SI for queries is an elegant solution to completely separate queries and update transactions while serializability is enforced.

In this chapter the metric to evaluate and compare protocols has been performance. However, the choice of the protocol also depends on the architecture of the underlying database and communication system. The question is how feasible it is to integrate each of these protocols into a specific database system. The next two chapters will discuss this issue in further detail.

6 Postgres-R

This chapter describes Postgres-R, an implementation of our replication framework. Building a working system has been motivated by a couple of important questions:

- Is it really possible to map our protocols to concrete algorithms? So far, the proposed protocols have been rather abstract using simple read and write operations. How can they be transformed to work on the complex operations encountered in real relational or object-relational systems (e.g. SQL statements)?
- How difficult is it to connect the replication mechanisms with an existing database and communication environment? How much must the underlying components change to support the new algorithms and are these changes complex?
- Finally, are the assumptions made by the simulation system realistic and can we really provide reasonable performance in a cluster environment?

There are two fundamental options to implement replica control. One possibility is to develop a replication tool as a middleware system that connects the different database systems. Such a tool could, e.g., be a component of a TP-monitor [BN97]. We have implemented a simple prototype of such a replication middleware [Rie99]. This middleware can connect any type of database system and the database systems themselves do not need to be changed. Hence, it allows for very general configurations and it is especially useful when database systems are used that do not provide replication. However, the approach has two major drawbacks. Firstly, performance is severely limited. All interactions with the database systems can only be through SQL statements (or corresponding interfaces) and little internal database optimizations are possible. Secondly, the replication layer must reimplement many features that are already provided within the database system, like concurrency control, logging, etc. This also requires the replication tool to maintain its own schema information. Having these components in two layers (database system and replication layer) is not only complex but has again negative impact on the performance. Böhm et. al [BGRS00] provide a performance evaluation of different middleware based replication architectures proving the inefficiency of most solutions.

The second option is to integrate the replication mechanism into the database system. As pointed out in Chapter 2, existing commercial database systems already provide sophisticated and efficient lazy replication mechanisms that take advantage of internal structures like logging, object versions, etc. Eager replication can only be an alternative to lazy solutions if it is integrated in the same way into the database engine.

Since the goal was to prove that eager replication is feasible, we decided to choose the second option as our main implementation. We used the database system PostgreSQL [Pos98] and extended it to build *Postgres-R* [Bau99b]. We chose PostgreSQL because it is one of the very few systems for which the source code is freely available and which has the full functionality of a relational database system (especially in regard to the ACID properties). The version of PostgreSQL that we use is, as most commercial systems, based on 2-phase locking. Due to the widespread use of 2-phase locking it seems natural to first implement replica control on top of this concurrency control method. Hence, we implemented the SER protocol. Since the simulation results have shown bad performance if SER is used for queries, we apply cursor stability for read-only transactions. We implemented the shadow copy approach since PostgreSQL supports triggers, constraints and interactive transactions and we wanted that these features can also be used in Postgres-R.

With respect to group communication, we could use any of the existing systems since our algorithms do not modify the communication primitives. Our current version of Postgres-R uses Ensemble [Hay98], the follow up system to Horus [vRBM96]. This decision has been rather pragmatic. Only very few systems are freely available. And of those that we have tested, Ensemble is the most stable, supports the highest number of nodes and has shown the best performance. The drawback of Ensemble is that it does not provide uniform reliable message delivery as an option. Hence, our performance tests could only be run with reliable delivery.

In the following, we will first present an overview of the underlying database system PostgreSQL. The description focuses on those parts and implementation details that are of interest for the integration of the replication tool. We will also comment on specifics and constraints of PostgreSQL and discuss how PostgreSQL differs from commercial systems. Then, we describe the architecture of Postgres-R along with the execution control, concurrency control, and message handling. The description depicts the general structure, points out some implementation details and discusses how the implementation would look like in other database systems. A last section summarizes our experiences and discusses shortly how the other protocols presented in Chapter 4 could be implemented.

6.1 An Overview of PostgreSQL

PostgreSQL is public domain software. It evolved from the Postgres prototype [SRH90]. Currently, PostgreSQL provides an extended subset of SQL and is a single machine database system running on Unix and Linux platforms. For our implementation, we use PostgreSQL version 6.4.2.

Architecture Figure 6.1 (a) depicts the general architecture of PostgreSQL. PostgreSQL is *process-based*, i.e., the units of execution are processes. The central process is the *postmaster* which listens for requests on a specific port. *Clients* wanting to access the database send a request to the postmaster. For each client, the postmaster creates a *backend* process and all further communication is done directly between backend and client using a two-way channel based on buffered sockets (implemented over TCP/IP or Unix domain sockets). PostgreSQL allows to limit the maximum number of parallel backends. When a given threshold is reached PostgreSQL will not admit any new clients. Limiting the maximum number of backends is necessary because process management is expensive. This mechanism also effectively limits the number of concurrent transactions thereby avoiding degradation in peak situations. The system maintains a *shared memory area* containing data accessible to all backends. This area includes the buffer pool, the lock table, and backend process information. Access to the shared memory is either controlled by semaphores or, if provided by the platform, by test-and-set instructions.

Most database systems have a similar architecture. Although many systems use threads instead of processes for executing transactions this does not change the conceptual structure. We assume, however, that commercial systems have a more efficient process/thread management than PostgreSQL. For instance, they usually always keep a pool of threads/processes active to serve user requests. This avoids frequent process creation/deletion. Additionally, many commercial systems maintain a whole suite of *background* processes (similar to the postmaster) that are running all time and are responsible for various jobs: administration of

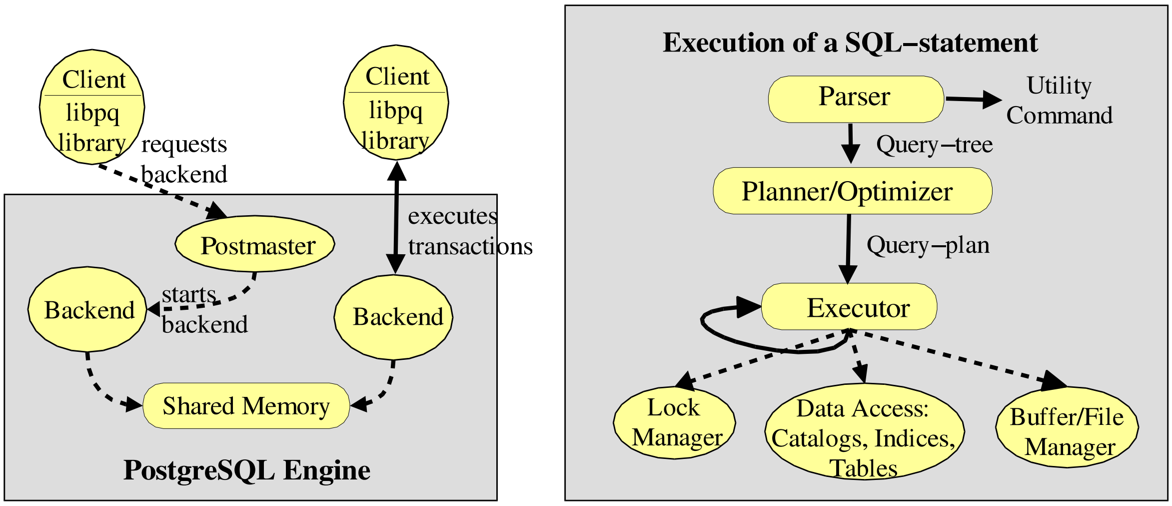

Figure 6.1: (a) Process structure and (b) execution control in PostgreSQL

the rollback segments, buffer management and asynchronous I/O, checkpointing, job dispatching, etc. Most of this functionality does not exist in PostgreSQL.

Interfaces PostgreSQL (as most other systems) offers a variety of interfaces: an interactive interface, several procedural languages (based on Perl, Tcl, SQL) to write stored procedures and triggers, a server programming interface, a C/C++ application programmer interface, a tcl package interface, embedded SQL in C, ODBC, JDBC and a Lisp programming interface. Most of these interfaces call an intermediate library `libpq` that represents the main interface between clients and backends. This means, the internal execution of a statement is identical for all external interfaces based on `libpq`. One of our goals has been to keep these interfaces available in Postgres-R without changing their implementation.

Transactions Clients can submit an arbitrary number of transactions (one at a time) until they disconnect. As default, each operation submitted by the client is treated as its own transaction. In order to bundle a set of operations into one transaction, PostgreSQL provides the usual `BEGIN` and `COMMIT` statements that have to enclose the operations. Each backend keeps track of the state of its connection to the client. The backend can be outside the scope of a transaction, it can be in the progress of executing a transaction, or it can be either aborting or committing a transaction.

Execution Control An SQL statement is executed in several steps (see Figure 6.1 (b)). We assume the execution to be similar in most relational databases. A backend receives a statement via data packets over the buffered socket connection to the client. The statement is first analyzed by the *parser*. If it is a utility command (anything but `select`, `insert update` or `delete`), it is processed by specific functions. Otherwise, the parser creates a “query-tree”. This query-tree is taken by the *planner/optimizer* which transforms it into a “query-plan” containing the operations needed to execute the statement. It does so by first creating all possible plans leading to the same result. For instance, assume an `update` statement and there exists an index on the search criteria. In order to find the qualifying tuples, either the index can be used or the corresponding relation can simply be scanned. The cost for the execution of each of these plans is estimated and the cheapest plan is chosen. The plan is represented as a set of nodes, arranged in a tree structure with a top-level node, and various sub-nodes as children. Finally, the *executor* takes the plan and executes it. It steps through it recursively calling itself to process the subplans, and retrieves tuples in the way represented by the plan. The executor makes use of the storage system which consists of a buffer manager, a lock manager, file management and some other components.

In the case of a `select` statement, each retrieved tuple is returned to the user. For complex queries (joins, sorts, etc.), the tuple might be stored in temporal relations and retrieved later for further processing from this temporal relation. In the case of the modifying statements `insert`, `update` and `delete` the executor performs the following steps:

1. Read next tuple according to the query-plan; if none left: exit.
2. Check existing constraints.
3. Fire before-triggers if necessary.
4. Perform the modification on the tuple.
5. Update indices if necessary.
6. Fire after-triggers if necessary.
7. Goto 1.

Multiversion System PostgreSQL is a multiversion system, i.e., each update invalidates the current physical version and creates a new version. To determine the valid version, each tuple has two additional fields which contain the identifiers of the *creating* and the *invalidating* transaction. A version is visible to a transaction if it has created the version itself or the creating transaction has already committed. Furthermore, for the tuple to be visible, the field for the invalidating transaction must be empty or the invalidating transaction is either still running or aborted. Thus, a transaction sees its own updates but not the updates of concurrent transactions. Updates trigger the creation of new entries in all relevant indices. Although these entries are readable to concurrent transactions, the corresponding tuples are not visible. To control the size of the tables, PostgreSQL provides a special garbage collector procedure to physically delete all invisible tuples.

It must be noted that PostgreSQL has a rather inefficient implementation of this multiversion scheme. The new tuple version is simply added to the end of the UNIX file containing the relation. The same happens to the new entries in the indexes. Furthermore, the old index entries are not invalidated. This means that for a tuple that has been updated x times, a primary index scan will find x index entries. For each of these entries the corresponding tuple version must be fetched to determine the valid version. In the same way, a normal scan has to read all the invalidated tuple versions before it will find the only valid tuple version. This can only be resolved by running the garbage collector. However, this requires an exclusive lock on each table and therefore cannot execute concurrently to normal transaction processing.

Most commercial databases are not multiversion systems but follow an update-in-place strategy where there is always only one version of a tuple in the database. An exception is Oracle. Although the tables only contain the latest version of a tuple, Oracle maintains special rollback segments storing enough information to reconstruct older versions that are accessed by read operations. We expect other systems to adapt similar strategies in the future to avoid interference between readers and writers.

Concurrency Control PostgreSQL 6.4.2 uses 2-phase-locking for concurrency control. Unfortunately, only relation level locking is implemented. Before the executor starts fetching tuples from a specific relation, the relation is locked. Most commercial systems provide lower granularities of locking, e.g., pages or tuples. Since relation level locking does not compare well with commercial systems, we have enhanced the locking scheme to allow for a very simple form of record level locking. We will discuss its implementation and further considerations about locking in one of the next sections.

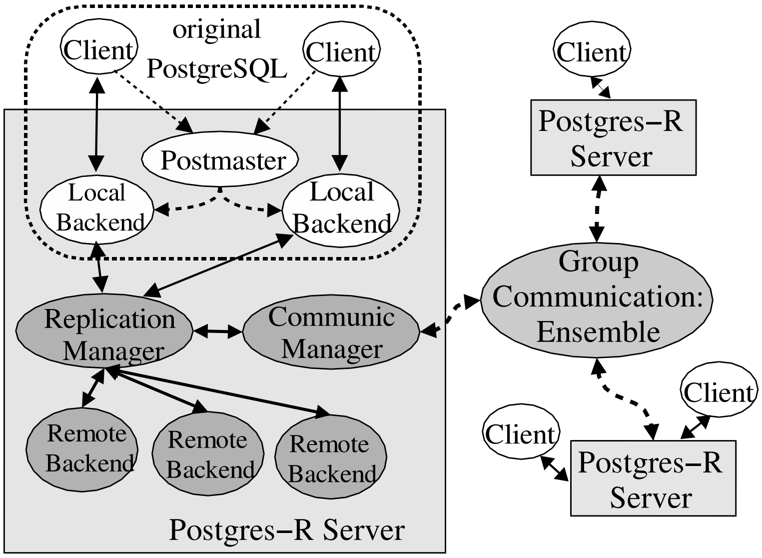

Figure 6.2: Architecture of Postgres-R

Recovery Typically, databases maintain a log containing undo and redo information for each operation executed in the system. The undo information is used in the case a transaction aborts. The redo information is used to reexecute operations of committed transactions if they are not reflected in the database after a failure.

In contrast, PostgreSQL implements a *no-undo / no-redo* recovery policy. Since it is a multiversion system, the valid version of a tuple is never overwritten, and hence, operations need not to be undone in the case of an abort. Furthermore, PostgreSQL flushes all dirty buffers to disk at the end of a transaction. Hence, the updates of all committed transactions are always on stable storage and no transaction must be redone in the case of failures. As a consequence, PostgreSQL does not write undo/redo log information but only a commit/abort log record for each terminating transaction.

6.2 The Architecture of Postgres-R

As shown in Figure 6.2, a Postgres-R configuration comprises several nodes (servers), each one of them running an instance of the Postgres-R database system. At each node, the process structure follows that of PostgreSQL. In the figure, all clear shapes represent original PostgreSQL modules. To implement replication, we have extended this structure and included several *background* processes per server that are, similar to the postmaster, created when a server is started, and alive until the server is shut down. These are the remote backends, the replication manager and the communication manager. The shadowed shapes of the Figure 6.2 depict this extension.

6.2.1 Process Structure

Backends A client may now connect to any of the Postgres-R servers. For each client, the postmaster creates a *local backend* process. The transactions of these clients are called the *local* transactions of the server, and the server is called the *local* node of the transactions or also the *owner* of the transactions. To handle *remote* transactions that have executed at remote sites and for which only the write sets are received, the server keeps a pool of *remote backend* processes. When the write set of a remote transaction arrives, it is handed over to one of these remote backends. Following the philosophy of PostgreSQL, we also allow to limit the number of parallel remote backends in order to control backend management and limit the number of concurrent transactions. This is done using two parameters. One parameter determines how many remote backends are created at server startup (the startup remote backend pool size). A second parameter determines the maximum number of parallel remote backends (maximum pool size). Upon arrival of a message, if all existing remote backends are busy and their number has not reached this threshold, a new remote backend is started, otherwise the transaction must wait. We have chosen to maintain a remote backend pool instead of creating a remote backend for each remote transaction because performance measurements have shown that backend creation and deletion are very expensive.

Replication Manager Control of the replication protocol takes place at the *replication manager* (created at server start-up). The replication manager is implemented as a message handling process. It receives messages from the local and remote backends and forwards write sets and decision messages via the communication manager to the other sites. It also receives the messages delivered by the communication manager and forwards them to the corresponding backends. The replication manager keeps track of the states of all the transactions running at the local server. This includes in which phases they are (read, send, lock or write phase), and which messages have been delivered so far. This is necessary in order to trigger the appropriate actions in the case of a failure. The replication manager maintains a two-way-channel implemented as buffered Unix sockets to each backend. The channel between a local backend and the replication manager is created when the backend sends its first transaction to the replication manager and closes when the client disconnects and the backend is killed. The channel between a remote backend and the replication manager is created once at backend startup time and maintained until Postgres-R is shut down. Similar to the process management, performance tests have shown that it is a very important engineering decision to keep the channels active over the lifetime of the backend process and not connect and disconnect for each new transaction since socket creation and deletion are extremely time consuming operations in Unix.

Communication Manager The only task of the communication manager is to provide a simple socket based interface between the replication manager and the group communication system (also the communication manager is created at server start-up). The communication managers of all servers are the members of the communication group and messages are multicast within this group. The separation between replication and communication manager allows us to hide the interface and characteristics of the group communication system. The replication manager maintains three one-way channels (again implemented as Unix sockets) to the communication system: a *broadcast channel* to send messages, a *total-order* channel to receive totally ordered write sets and a *no-order* channel to listen for decision messages from the communication system. There are two receiving channels because we want decision messages to be received at any time, while reception of totally ordered write sets will be blocked in certain phases.

6.2.2 Implementation Issues

For the clients, the changes in architecture are completely transparent. The interface to the database system is unchanged and only the local backends are visible to clients. Application programs can be written and used as with PostgreSQL.

The replication manager and the communication manager are implemented independently of PostgreSQL, and we expect that these modules could be used in other systems without significant conceptual changes. We have integrated them as *background processes* into PostgreSQL. The same holds for the remote backends. Adding replication components as new background processes seems a natural extension of the typical architecture of database systems.

Our design favors modularity (by using various processes) and structured flow control (by using buffered sockets for inter-process communication). The disadvantage of this approach is that each message exchange consists of several process switches (backend, replication manager and communication manager) and memory copies (between the buffered sockets). General guidelines for an efficient inter-process communication, however, recommend to avoid process switches and memory copies as much as possible. To change our design according to these guidelines would not be extraordinarily difficult: the functionality of the communication manager could be integrated into the replication manager reducing context switches. Memory copies could be reduced by using shared memory or by allowing backends to send messages directly to other sites instead of forwarding them to the replication manager. We have not implemented these alternatives because they reduce modularity, and leave the system with an unclear distribution of tasks and responsibility, and a more difficult handling of failures. Furthermore, the observed performance overhead of our implementation has been so small compared to other steps in the execution of the transaction that we considered it acceptable.

6.3 Execution Control

The best way to understand how transactions are processed is to follow the different steps of their execution: read phase, send phase, lock phase, write phase (in Postgres-R, the termination phase is included in the write phase). For each transaction, its current phase is recorded at the backend and at the replication manager. Figure 6.3 shows an example execution at the local server (top) and a remote server (bottom) depicting the actions of the backend and the replication manager and their interaction. We have omitted the communication manager since it only has forwarding functionality. The figure represents only one possible execution flow (the transaction commits and all messages arrive in the order expected). In order to cover all possible cases, Figure 6.4 shows the complete state machine of a transaction at the local replication manager (top) and a remote replication manager (bottom).

6.3.1 Local Transactions

Local Read Phase and Send Phase For local transactions, as long as they are in their read phase, they remain within their corresponding local backend and are still unknown at the replication manager. Once the local backend finishes the execution (over shadow copies), it sends the write set of the transaction to the replication manager and the transaction enters its send phase. The replication manager then multicasts the write set to all sites. When a write set arrives at a site, the corresponding replication manager checks whether the write set corresponds to a local or to a remote transaction. This can be done because the write set message includes the host name and a transaction identifier.

Local Lock Phase For write sets of local transactions, the replication manager notifies the corresponding local backend and the transaction enters its *lock phase*. In order to perform the lock phase atomically, the local backend acquires a latch on the lock table and keeps it until all locks are enqueued in the lock

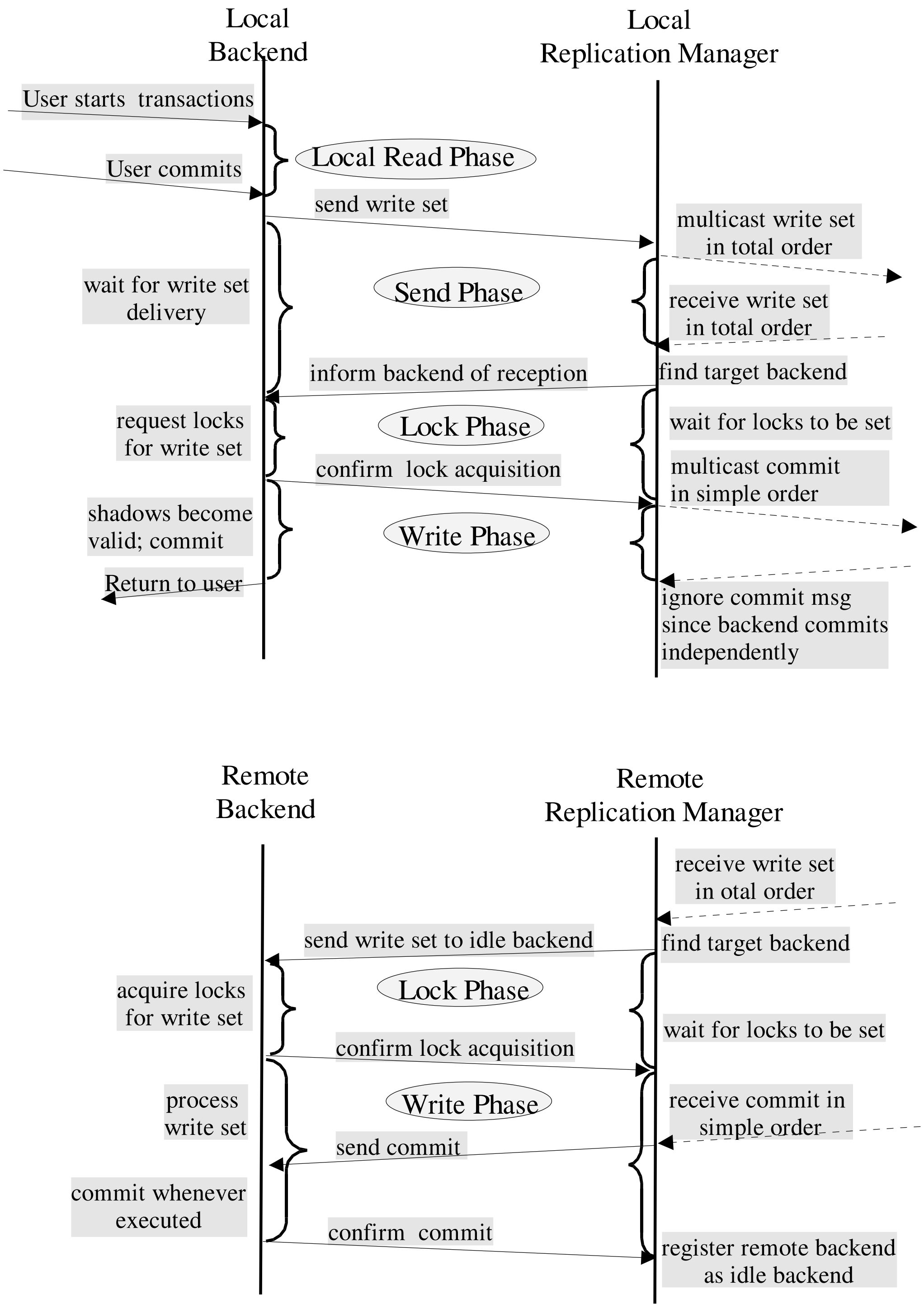

Figure 6.3: Example of the execution of a transaction at the local (top) and a remote (bottom) server

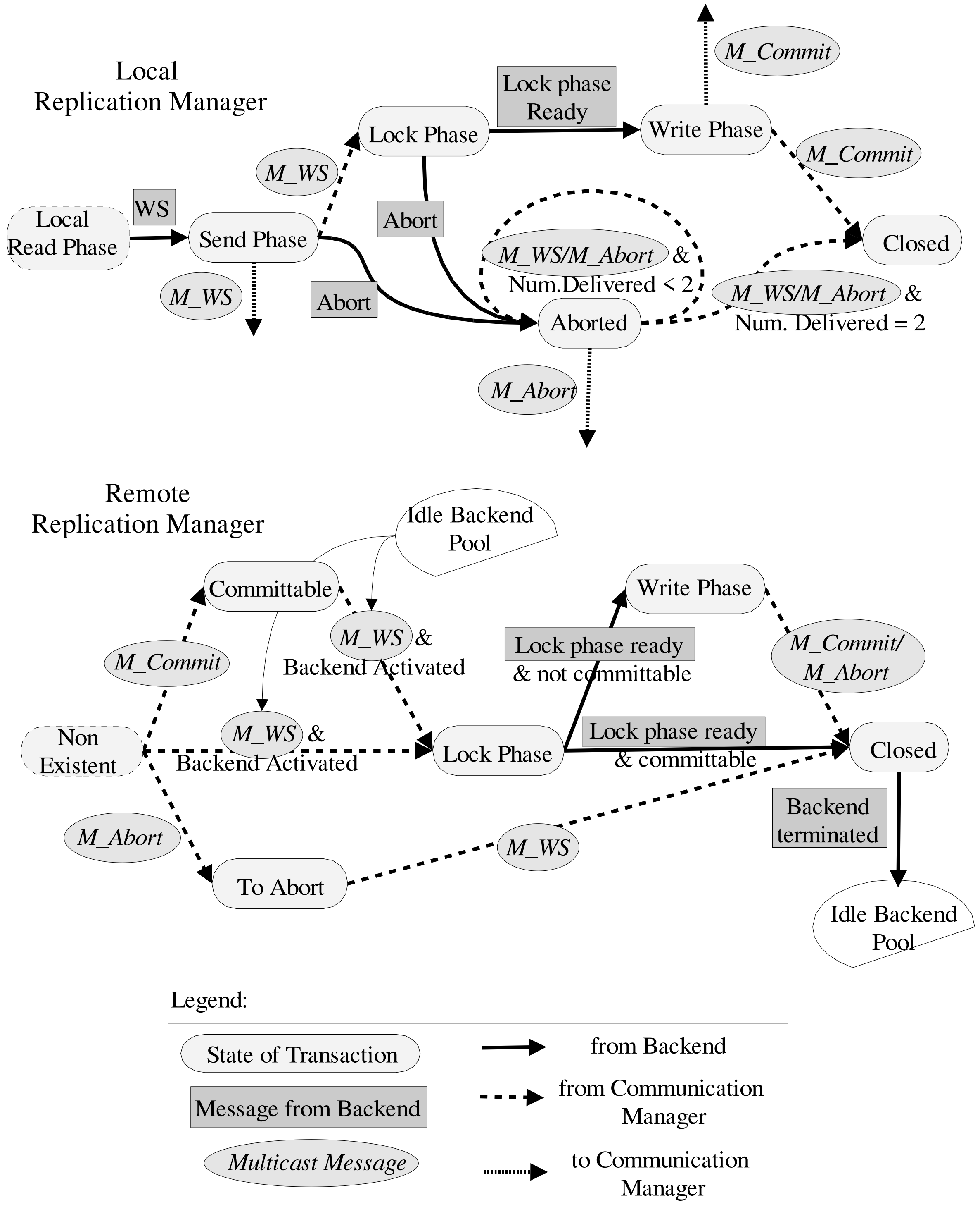

Figure 6.4: State machine of a transaction at the local (top) and a remote (bottom) replication manager

table. Additionally, the replication manager stops listening on the total-order channel until the backend sends a confirmation that all necessary locks have been requested. This guarantees that the lock phases of concurrent transactions are executed in the same order in which the write sets have been delivered by the group communication system. Note, that the lock phase is different to usual locking schemes in that several locks are requested in an atomic step. As a result a transaction can have more than one lock waiting.

Local Write Phase When the replication manager receives the confirmation from the backend that the locks have been requested, it multicasts a commit message. The transaction then enters its write phase and whenever a lock is granted the shadow copy becomes the valid version of the data item. At the backend, the transaction is committed once all locks have been granted and the database contains the valid data versions. At the replication manager, once it receives the commit message, information about the transaction is removed.

Abort of Local Transactions Until the time when the local backend has acquired the latch on the lock table to start the lock phase, a local transaction can be aborted due to a read/write or a RIW/write conflict. This happens when a transaction tries to set a write lock (during its lock phase) and finds a read/RIW lock from a transaction that has not yet passed its lock phase, i.e., it is still in its read or send phase. In this case, the writing transaction sets an abort flag for the transaction holding the read/RIW lock and enqueues its own write lock request. Local transactions check their abort flag regularly and if they find it set, they abort. The last time a transaction checks is at the begin of the lock phase directly after acquiring the lock table's latch. After that it cannot be aborted anymore. If a local backend finds out that its transaction has to be aborted and the write set has already been sent, it sends an abort message to the replication manager. The replication manager then multicasts this message and the transaction is considered aborted. Note that there is some asynchrony between the replication manager and the backend when the transaction enters a certain phase. For the replication manager the lock phase starts when it puts the confirmation of the arrival of the write set into the socket to the backend. For the backend, the lock phase only starts, when it has successfully acquired the latch on the lock table. Hence, the replication manager might receive an abort message from a transaction from which it thinks it is already in its lock phase. The replication manager deletes the information about the transaction only when it has received both the write set and the abort message.

6.3.2 Remote Transactions

For remote transactions, write and decision messages might arrive in any order at the replication manager. If the write set arrives first, the replication manager passes this information to an idle remote backend and proceeds like before. The remote backend will first request the locks (lock phase). Here, the atomic request of several locks does not only mean that the transaction can have more than one lock waiting (as with local transactions) but it can also have more than one lock granted without the corresponding operation on the object being executed. When all locks are requested, the remote backend confirms this to the replication manager, and executes the operations or applies the updates (write phase) once the corresponding locks are granted. However, it will wait to terminate the transaction until the replication manager receives the decision message from the local site and forwards it to the remote backend. Once a remote backend has terminated a transaction, it sends a confirmation to the replication manager, which will add the backend to the pool of available remote backends. In the case there is no idle backend, the replication manager blocks the total-order channel and only listens on the decision channel until a remote backend notifies the replication manager about the termination of its transaction.

If the decision message arrives first, the replication manager registers the outcome of the transaction. When the outcome is abort the replication manager simply deletes the information of the transaction whenever the write set is received. No backend is activated in this case. When the decision is commit, upon reception of the write set the replication manager forwards both write set and commit to an idle backend and proceeds as before.

6.3.3 Failures

In case of failures, Postgres-R implements the semantics for the SER protocol with reliable delivery described in Section 4.3.2. When sites fail, the group communication system informs the replication managers of available sites via a view change message. In this case, the replication managers force the abort of all pending transactions (remote transactions of failed nodes with missing decision messages) and transaction processing can continue. In the case of a network partition, transaction processing is stopped in all minority views. This guarantees data consistency on all available sites. For more details see [Bac99].

6.3.4 Implementation Issues

Local backends Local backends required a couple of changes at specific points in the code and some new code had to be added. Some of these changes have been very specific to the constraints given by PostgreSQL and might look different or not even be needed in other systems.

- **RIW locks:** During the read phase a transaction does not request write locks but RIW locks. The extension of the lock management to support RIW locks has been quite straightforward. In fact, most systems provide an entire hierarchy of lock types (different granularities, different modes) and extending it should generally not be difficult. It is also easy to automatically control which lock has to be acquired at what time. The decision depends only on whether a table is replicated and in which phase a transaction is.
- **Write set:** Whenever a local transaction performs a write operation, the corresponding entries in the write set are prepared. While we had to add a complete new module responsible for message marshalling and unmarshalling, systems that maintain a log can create the write set by extracting the information from the redo log which will reduce the programming overhead. This is already done in commercial systems supporting lazy replication.
  Furthermore, we had to change the executor code of PostgreSQL to include the activation of message marshalling. Alternatively, systems like Oracle, that have a very powerful trigger system, might be able to activate the write set creation by triggers without changing the underlying execution control. Changing the executor code can also be avoided, if the write set can be reconstructed from a redo log. With this, marshalling can be done directly before the message is sent.
- **Send, lock and write phase:** The original commit has now been extended to also include send, lock and write phase. The local site sends the write set, receives the confirmation about its delivery, requests all write locks, and sends the commit message. We consider the programming effort involved in extending the centralized commit comparable to implementing a 2-phase commit protocol in regard to its complexity and where execution control has to be changed.
- **Local aborts:** The abort of a local transaction T can be induced at any time during its read and send phase by a transaction T' in its lock phase. However, T' cannot simply send a “kill” signal to T since T can be in some critical section (holding pins on buffers, being in the middle of I/O etc.). Since it

was difficult to determine at which points a reading transaction T can safely be aborted, T' only sets an abort flag in T. T then checks this flag at specific time points at which it is safe to abort.
In general, the abort implementation will be very system dependent, Nevertheless, since most systems already provide a careful handling of aborts in many occasions, the inclusion of another possible abort situation should be feasible in any case.

- **Shadow Copy:** The shadow copy approach could be implemented in a straightforward way. The multiversion implementation of PostgreSQL could be directly used as a shadow copy scheme without any changes to the code, since each update produces a new version of the tuple and these versions are only seen after the commit. The implementation of the shadow copy approach should be equally feasible in any database with a built-in multiversion system (e.g., Oracle). As noted in Chapter 4, if a system does not support multiple versions, the write operations could directly be performed on the valid data versions if the correct locks are set.

Remote backends Although remote backends are a new component, most of the code is based on PostgreSQL functionality. A remote backend is started in the same way as a local backend to keep it under the recovery control of the postmaster. Also the basic transaction framework is executed as in an original backend process. If the write set contains an SQL statement it uses the same functions as the local backend to execute it. In the case the physical changes are sent, we use PostgreSQL index functions to directly receive the tuples and perform the changes. Since we only used but did not change existing PostgreSQL code, we consider the remote backend implementation as very modular. It should be possible to integrate the remote backend in the same way in any other system.

6.4 Locking

The concurrency control is the component that has been most affected by the replica control system. In a first step, we had to enhance the locking mechanism of PostgreSQL to be able to request several locks in one step and to allow several locks of the same transaction to wait at the same time.

Tuple Level Locking Furthermore, since PostgreSQL only supports relation level locking, we have implemented a simple tuple level locking scheme using logical read, RIW and write locks. The lock key is a combination of the relation identifier and the primary key of the tuple. We would like to note that this scheme has only be implemented for performance comparisons and, as it is, is not completely practical in a real setting. `Select` statements still use relation level locks, however, one can choose to use short or long locks. For `insert`, `update` and `delete`, we require the tuples to be accessed to be easily determined by parsing the SQL-statement (search criteria only on the primary key). With this, it is possible to request the RIW locks in advance before performing the executor steps and accessing the tuples. When the write set is sent, it contains the primary keys for all modified tuples. Hence, all write locks are known when the write set is delivered and can be accessed in a single atomic step during the lock phase.

Alternatives The tuple level scheme implemented has severe restrictions. In real settings, `insert`, `update` and `delete` statements can also have arbitrary search criteria in their `where` clauses. Hence, the tuples can often not be identified before executing the operation. Instead, tuples are fetched and examined. When they fulfill the search criteria, they are locked. This means, however, that two transactions can fetch (read) the same tuple concurrently while only one will be successful acquiring the RIW lock. In PostgreSQL, the successful transaction will create a new and invalidate the old version of the tuple. As a result, the waiting transaction points to an invalid version. PostgreSQL avoids this by acquiring relation level locks. As an alternative and in order to allow tuple level locks, update operations could acquire short exclusive locks (latches) on physical pages before the tuple is read. Such a lock can be released once all qualifying tuples on the page have been successfully modified. This, however, has to be implemented in the buffer management with the knowledge of how tuples are exactly stored on pages and files – a section of PostgreSQL that we decided not to change.

Locking in other Database Systems Most commercial systems do support tuple level locking and the restrictions above do not apply. Still, problems might arise since our approach requires *logical locks*. A lock is logical if it is identified by the identifier of the data item. As described above, in a relational system this is, e.g., a combination of the identifier of the relation and the key of the tuple. In object-oriented systems this could be a combination of object and instance identifier.

While object-oriented systems usually support some form of logical locks, many relational systems are based on *physical locks*. Here, a tuple lock is usually a physical tuple identifier `TID`. A `TID` consists of the identifier of a physical page and an identifier of the section of the page in which the tuple is stored (indirections are possible). Physical locks are problematic in distributed systems, since the `TID`'s of the same tuple will be different at the different sites. Additionally, since the `TID` is a pointer to a physical address, it can only be determined when accessing the tuple (or, if an index is used, when the corresponding entry in the index has been found). In our protocol, however, remote sites need to know the lock identifiers before accessing the tuples. This problem can only be solved by using logical locks that are the same for all sites. How difficult it is to integrate logical locks into a system based on physical locks remains to be examined.

Index Locking Locks on index structures need further consideration. Most of these locks are usually short locks not following the 2-phase-locking requirement and hence, can be acquired at any time, even during the write phase of a transaction. In B-trees, for instance, while searching for an entry to be updated, PostgreSQL searches along a path in the B-tree, locking and unlocking (short read locks) individual pages until the entry is found. When the entry is found, the short read lock on the corresponding page is upgraded to a write lock. Two transactions can follow this procedure at the same time and deadlock when they try to upgrade the lock. In PostgreSQL, such deadlocks occur frequently because each update operation creates a new entry in the primary key index. However, since indices are also used during the write phase, remote transactions could also be involved in such deadlocks. To avoid this, Postgres-R immediately acquires write locks on index pages in the case of update operations. An alternative would be to not abort the transaction upon a deadlock on upgrading index locks. Instead only the granted index locks could be released and the index search restarted. This does not violate correctness, since index locks are short in any case. Deadlocks due to upgrading locks are a general problem in databases and we believe that many systems have installed mechanisms like above to avoid them.

6.5 The Write Set

Creating and sending the write set plays a crucial role in our protocols. If it is not carefully done, it can have a severe impact on performance. In particular, the write set must be packed at the local site, transferred over the network, and unpacked and applied at the remote sites.

Sending the SQL Statements An easy way to transfer write operations is to simply send all modifying SQL statements (i.e., `update`, `delete` and `insert` statements; `select` statements are still only

1 operation updating	Message Size in Bytes		Execution time in ms				
			Not	Local Site		Remote Site	
	SQL	physical	replicated	SQL	physical	SQL	physical
1 tuple	123	105	7	7	7	7	1
50 tuples	125	3623	125	125	140	125	40

Table 6.1: The write set

executed locally). Note, that this would be the only possibility in the case of deferred writes. With this, messages are small but remote sites have considerable work since they need to parse the SQL statement and execute the entire operation. Additionally, care has to be taken if the SQL statement contains implicit reads. In this case, read locks must be requested during the lock phase for data that will be read but not written. These read locks may not lead to aborts upon conflicts with other write operations.

Sending the Physical Changes Using the shadow copy approach, updates are already executed during the read phase, and we can directly send the physical changes of the individual tuples to the remote sites. In the case of an `update` statement, we send for each modified tuple the primary key value and the new values of those attributes that have been updated. In the case of `insert`, we send the complete new tuples, in the case of a `delete`, we only send the primary key values. At the remote site, the specific tuples are accessed via the primary key index. With this, remote sites are much faster installing updates, but messages can become quite large if many tuples are accessed or the modified attributes are large. This can lead to severe latencies and buffer management problems in the communication system.

Mixture Since both approaches have their advantages and disadvantages, Postgres-R uses a mixture of sending SQL statements and the physical changes. We allow the specification of a tuple threshold. If a write operation updates less than the given threshold of tuples, the physical changes are transferred. Otherwise, the SQL statement is sent. We allow a write set to be mixed, consisting on the one part of SQL statements on the other part of record information.

In all cases, and in order to preserve dependencies, the write set is *ordered* by adding tags to the primary key values or the SQL statements. These tags indicate in which order they have to be executed at the remote sites. This order is not a total order but allows parallel execution whenever there are no dependency constraints. For further details see [Bau99b].

Evaluation The performance differences between the two approaches can be seen in Table 6.1. We have run two tests. In the first test, a write set contains a single operation updating one tuple. The index on the primary key can be used to find the tuple. In the second test, there is one operation updating 50 tuples. This statement performs a scan through a table consisting of 1000 tuples. In both cases, two tuple attributes are modified. The table indicates the message size and the execution time for the operation at the local and the remote site. We also indicate the execution time in a non-replicated system.

Regarding message size, in the 1-tuple case, there are no significant differences between sending statements or the physical updates. However, with 50 tuples, the message containing the physical changes is quite big and might lead to problems in the communication system. Regarding execution time, if only one tuple is updated or if the SQL statement is sent, the overhead for message packing at the local site is not visible and execution takes as long as in the non-replicated case. But even if the local site must pack the physical updates of 50 tuples into the write set, the overhead is not very high. The most visible difference, however, is how much faster a remote site can apply the physical updates in comparison to executing the SQL statement.

Considering that the overhead of creating the write set at the local site occurs only once while applying the updates occurs at many remote sites sending the physical changes reduces the resource consumption considerably. How this effect reduces the overall response time has already been briefly shown in the simulation results of the previous chapter and will become even more visible in the experiments of the next chapter. As a conclusion, sending the physical updates should always be given preference, as long as message size is not the limiting factor.

6.6 Discussion

The implementation of Postgres-R has provided interesting insights into the practical issues of integrating eager update everywhere replication into an existing database system.

The Advantages of PostgreSQL In some cases, the specifics of PostgreSQL have been helpful for our purposes. With PostgreSQL being a multiversion system, it was rather straightforward to implement a shadow copy approach. Furthermore, the rather simple, process-based structure of PostgreSQL helped to speed-up programming. However, although working with a thread-based system would have probably been more time-consuming we believe it will not have been different from a conceptual point of view.

The Restrictions of PostgreSQL On the other hand, we believe that some other parts will be easier to implement in other systems. Write set marshalling will probably be better supported in a system that maintains redo-logs. A module implementing 2-phase commit might have been helpful as a starting point to implement send, lock, and write phase. The biggest restriction of PostgreSQL, however, has been the table based locking management. It forced us to invest considerable programming effort in the concurrency control management and left us with a suboptimal solution. Other systems do not have this restriction. However, replica control will always be sensitive to the underlying concurrency control mechanisms.

Different Levels of Isolation So far, Postgres-R supports only the SER protocol with some CS extension (since one can choose to either use short or long locks for `select` statements). A proper implementation of the CS protocol would only require two extensions: a primitive must be provided which allows to choose between SER or CS (e.g. a statement like `` ``set isolation level CS'' ``). Furthermore, `select` statements must be extended so that they can contain a `for update` clause. Then, the database automatically uses long read locks in the case the clause is set, otherwise locks are released after the execution of the operation. Many systems already provide such primitives so that the CS protocol would come for free once SER is implemented.

The SI protocol and the hybrid protocol using snapshot isolation for queries need concurrency control support for snapshot isolation. If the underlying database system does not support it, it must be added. For systems that provide snapshot isolation, we believe that integrating the SI replica control protocol will not be more complicated than integrating the SER protocol into a 2-phase locking component. All the other parts of the replication tool (replication manager, communication manager, write set module) would need little to no modification.

Different Levels of Fault-Tolerance The level of fault-tolerance is independent of the rest of the replica control system but only depends on the underlying group communication system. The database system behaves nearly the same whether reliable or uniform reliable delivery is used. Only in the failure case and during recovery special steps have to be taken in the case of reliable delivery. We discuss them in Chapter 8. Unfortunately, Ensemble, the group communication system used, only supports reliable delivery. We see this as a considerable restriction that we hope to revoke as soon as we have access to a communication system supporting reasonable efficient and stable uniform reliable delivery.

Deferred Writes vs. Shadow Copies In order to compare the programming effort we have also implemented the deferred writes approach. Implementing deferred writes has been rather simple. Any `insert`, `update` or `delete` statement is packed into the write set message but not executed. Upon reception of the write set the corresponding locks are requested and the write operations are executed on all sites (including the local site). The deferred writes approach can generally not use tuple level locking if the update statements have implicit read operations in their `where` clauses, since all write locks must be acquired before the operations are executed. The ideal lock granularity would be predicate locks that lock exactly the sets of tuples that are defined by the `where` clauses. Some but not many relational systems support easy forms of predicate locks [Moh90a, GR93]. Since predicate locks are not supported in PostgreSQL, we set relation level locks.

The implementation overhead of both approaches was not considerably different but executing the write operations during the local read phase has proven to have many more advantages. Besides the richer functionality (handling of write/read dependencies, triggers and constraints) it has performance advantages in two regards. First, it allows for more concurrency than the deferred update approach since Postgres-R does not support predicate locking. Second, we can send the physical updates reducing resource consumption considerably. These advantages will be effective in most relational database systems. How far they also hold for object-oriented systems remains to be analyzed.

Summary By integrating our replication approach into PostgreSQL we have proven that our replication approach is feasible in practice. Postgres-R maintains a modular structure and its architecture is very similar to the general architecture of most relational database systems. Most of the features necessary to support replication could be *added* to the system in separate modules, and only *few changes* had to be made to existing modules. Thus, we believe that our approach can be integrated in a similar way into many other relational database systems.

7 Evaluation of Postgres-R

After describing the implementation of the protocols, there remains the question of the overall performance. From a theoretical point of view, the protocols seem to avoid many of the problems associated with eager replication. It needs to be tested whether they can really provide the performance needed in cluster databases. For this purpose, we have analyzed the performance of Postgres-R for a variety of configurations and workloads. In a first experiment we compare the relative performance of traditional eager replication and Postgres-R for small system sizes and workloads. Then, we perform a suite of experiments testing the full capacity of Postgres-R and its limits. In particular, we test the system for high throughput rates, many clients, high conflict rates, and large system sizes. We analyze how Postgres-R behaves if nodes are differently loaded, and how much the performance of Postgres-R is influenced by the capacity of the communication system. Finally, we test the system with a workload representative for typical cluster applications. With this suite of experiments we try to cover as many scenarios as possible and to provide a representative performance analysis of Postgres-R in a cluster configuration.

7.1 General Configuration

Replica Control The tests in this chapter only discuss the shadow copy approach. Since the deferred write approach of Postgres-R acquires relation level locks concurrency is limited to such an extend that we were not able to achieve acceptable performance. Furthermore, results are only shown for the approach in which remote servers apply the physical updates.

Database Schema and Transaction Types In all our experiments, the database consists of 10 tables each containing 1000 tuples. We did not consider larger databases since this will only reduce the conflict profile. We also concentrated on OLTP loads, that is, we tested the system with many short transactions. Each table has the same schema: two integers (one being the primary key `table-id`, the other denoted below as `attr1`), one 50-character string (`attr2`), one float (`attr3`) and one date (`attr4`) attribute. For each table there exists one index for the primary key.

Update transactions consist of a number of operations of the type

```
update table-name
set attr1='x', attr2=attr2+4
where table-id=y
```

where x is randomly chosen text and y is a randomly chosen number between 1 and 1000. The relevant tuple is found by searching the index on the primary key. Queries consist of one operation requiring to scan an entire table and are of the form

```
select avg(attr3), sum(attr3) from table-name
```

Clients Transactions are submitted by clients which are evenly distributed among the servers. Efficient client management is crucial for performance. PostgreSQL has a rather inefficient client management since it creates a new process for each new client and the costs for socket administration arise whenever clients connect and disconnect. Hence, we have decided to keep the number of clients constant during an experiment, whereby each client connects once to its local server and submits transactions until the end of the test.

Workload The interarrival time between the submissions of two consecutive transactions is exponentially distributed. The submission rate (also referred to as workload) varies throughout the experiments and is determined by the number of clients (also called multiprogramming level) and the mean interarrival rate for each client. It is denoted by the number of transactions submitted per second (tps). Except in a few experiments where the system was saturated with transactions, the system throughput is equal to the submission rate. Whenever a transaction is aborted, the client resubmits it immediately.

Hardware Configuration For all but the first experiment where we had to use a different platform, we used a cluster of 15 Unix workstations (SUN Ultra 10, 333 Mhz UltraSPARC-IIi CPU with 2MB cache, 256 MB main memory, 9GB IDE disk), connected by a switched full-duplex Fast Ethernet network. We did not have exclusive access to the cluster but run the experiments when the interactive workload was generally low.

PostgreSQL Configuration PostgreSQL forces all dirty pages to disk at the end of the transaction to avoid redo recovery. Due to this *force* strategy, response times are very poor. A transaction consisting of 5 tuple updates in a single server system takes about 400 ms, whereby 350 ms are devoted to a flush of the buffer. PostgreSQL does not only flush pages that were modified by the committing transaction but all dirty pages. This makes it difficult to compare with commercial systems which only flush redo logs sequentially to disk. To allow us to use a more “realistic” setting we used the no-flush option offered by PostgreSQL. With this option, nothing is forced to disk. This, of course, violates the ACID properties. However, the measured response times could be better compared to standard database systems.

Set-up of the Test Runs Table 7.1 summarizes the parameters of the different experiments. As performance indicator, we analyze the response time of local transactions, i.e., the time from which the client submits a transaction until the client receives the commit. For comparison: in an optimal system (single user, single node), an update transaction with 10 operations takes approximately 65 ms. An update transaction with 1 operation takes 9 ms. A query requires 50 ms.

All tests were run until the results were stable or at least 4000 transactions have been executed. There is one major reason why we did not use the confidence interval method. Due to the implementation of the multiversion system (see Section 6.1) the physical size of the database grows continuously. Data access times grow accordingly since the primary index contains more and more entries for each tuple and all versions of a tuple have to be examined to find the valid version. Interestingly, the resulting response time increase has not been visible at small loads. When the system was loaded, however, this effect was significant. To achieve comparable results we reinstalled the database before each test run and for each test suite we run the same amount of transactions.

Experiments	2PL	WL I	WL II	WL III	Conflicts	Comm.	Queries
Database Size	10 tables of 1000 tuples each						
Tuple Size	appr. 100 Bytes						
Hot Spot	0%	0%	0%	0%	varying	0%	0%
# of Servers	1-5	10	1-15	10	10	1-15	1-15
% of Upd. Txn.	100%	100%	100%	100%	100%	100%	varying
# Oper. in Update Txn.	5	10	10	10	10	1	10
# Oper. in Query	-	-	-	-	-	-	Scan of 1 table
# of Clients	1-8	1-30	5-30	20	20	20	2 per server
Submission rate in tps in the entire system	10	10-45	10-50	10-40	10-40	40-200	15-225

Table 7.1: Parameters

7.2 Distributed 2-phase Locking (2PL) vs. Postgres-R

In this first experiment we compare distributed 2-phase locking with Postgres-R. To do so we use one of the commercially available implementation of eager replication, that is, the one provided by Oracle Synchronous Replication [Ora97]. This experiment allows us to check the limitations of traditional eager replication. Furthermore, it provides us with a first impression of the performance of Postgres-R when compared with that of a commercial product.

The workload consists of only update transactions, with 5 operations each. While the number of clients was varied from 1 to 8, the workload for all test runs was fixed to 10 tps for the entire system (i.e. having 1 client, this client submits a transaction each 100 ms, having 5 clients, each client submits a transaction each 500 ms). The number of replicated nodes was varied from 1 (no replication) to 5.

Standard Distributed Locking In Oracle Synchronous Replication, updates first obtain a lock on the local tuple and then an "after row" trigger is used to synchronously propagate the changes and to lock the corresponding remote copies of the tuple. At the end of the transaction a 2-phase commit takes place and the locks are released. The experiment was conducted with 5 Oracle instances installed on PCs (266MHz and 128MB main memory, two local disk [4GB IDE, 4GB SCSI]) connected by 100Mbit Fast Ethernet (switched full-duplex). We did not tune any of Oracle's startup parameters but used the default values. Only the timeout for distributed transactions was set to the minimum value of 1 second (the default value was 60 seconds).

The results of the experiment are shown in Figures 7.1 (response times), 7.2 (throughput), and 7.3 (abort rate). Clearly, as the number of replicated nodes (servers) increases, the response times increase and the throughput decreases. Using one server, the system was running in non-replicated mode. In this case, transaction response times are below 100 ms and increase only slightly with the number of concurrent clients; the conflict rate is small, no deadlocks occur and the CPU is far of being saturated. Once the

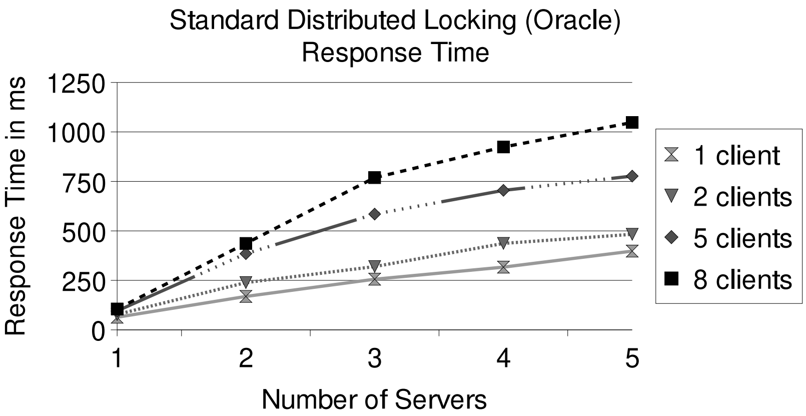

Figure 7.1: 2PL: Response time of distributed 2PL (Oracle) at a submission rate of 10 tps

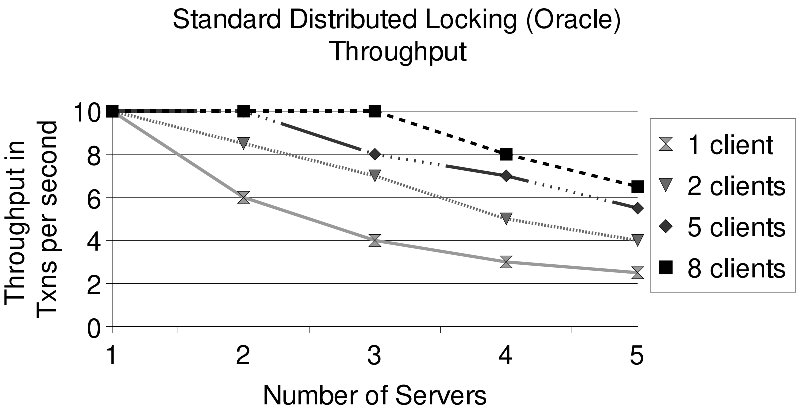

Figure 7.2: 2PL: Throughput of distributed 2PL (Oracle) at a submission rate of 10 tps

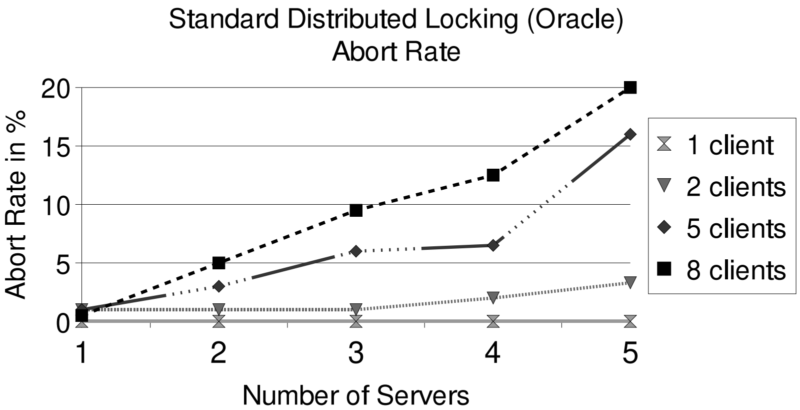

Figure 7.3: 2PL: Abort rate of distributed 2PL (Oracle) at a submission rate of 10 tps

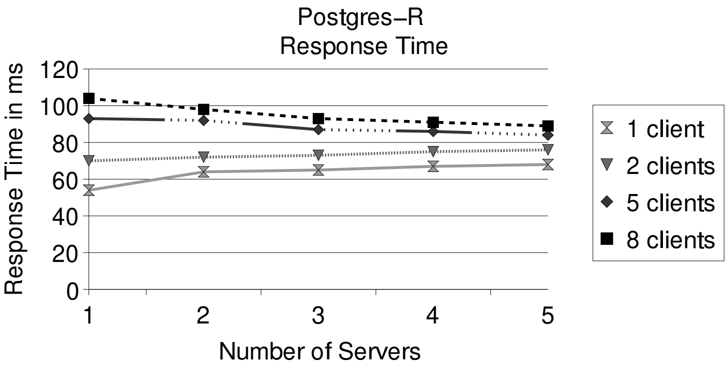

Figure 7.4: 2PL: Response time of Postgres-R at a submission rate of 10 tps

system is replicated, performance degrades very fast. Using only one client (i.e., no concurrency at all), response time increases steadily when new servers are added to the system and at five servers response time is more than 6 times as high as in a non-replicated system. Already with 2 servers the throughput of 10 tps could not be reached because response time was longer than the transaction interarrival time. Using higher multiprogramming levels result in even higher response times. The sharp bends in the response time curves show the point where the required throughput is not reached. Abort rates increase with the number of clients and the number of nodes. In a configuration of 3 servers and 5 clients, the abort rate is around 6%. Using 5 servers and 5 clients, it is 16%. As already noted in Chapter 5, it has been difficult to set the appropriate timeout interval. Obviously, such a mechanism does not scale beyond a few nodes (Oracle manuals warn users that this is, in fact, the case). We believe that these results are expected and typical of traditional implementations of eager replication. It is this type of implementations that has been analyzed and criticized by Gray et al. [GHOS96].

Postgres-R As a first comparison, not with Oracle, but with traditional eager protocols, we did a similar experiment with Postgres-R. The experiment was performed on 5 Unix workstations (SUN Ultra 5, 269 MHz, 192MB main memory and one local 1GB disk). Unfortunately, we could not use the same hardware platform as for Oracle. Also note that Oracle and PostgreSQL differ in many aspects and we used some special optimizations in Postgres-R (remember that the tests are done without flushing the log to disk). This means, the comparisons can only be relative. However, the test gives an idea of whether Postgres-R suffers from similar limitations as Oracle eager replication.

Figure 7.4 shows the response time for this experiment. Before discussing the replicated configuration we would like to point out that the non-replicated PostgreSQL (1 server) has considerable performance differences for different number of clients (multiprogramming levels). This shows that PostgreSQL – unlike Oracle – has a rather inefficient client management. For 8 clients response times are more than double as long as for 1 client (although the throughput is the same).

Looking at the replicated system, the performance of Postgres-R proves to be stable as the number of replicated nodes increases and no degradation takes place. The figure shows that the response times are stable independently of the number of servers. The throughput of 10 tps is achieved for all test runs and abort rates are clearly below 5% for all experiments (not explicitly depicted in figures). None of the resources is ever saturated. Looking at a configuration with 1 client we can see how response time increases by

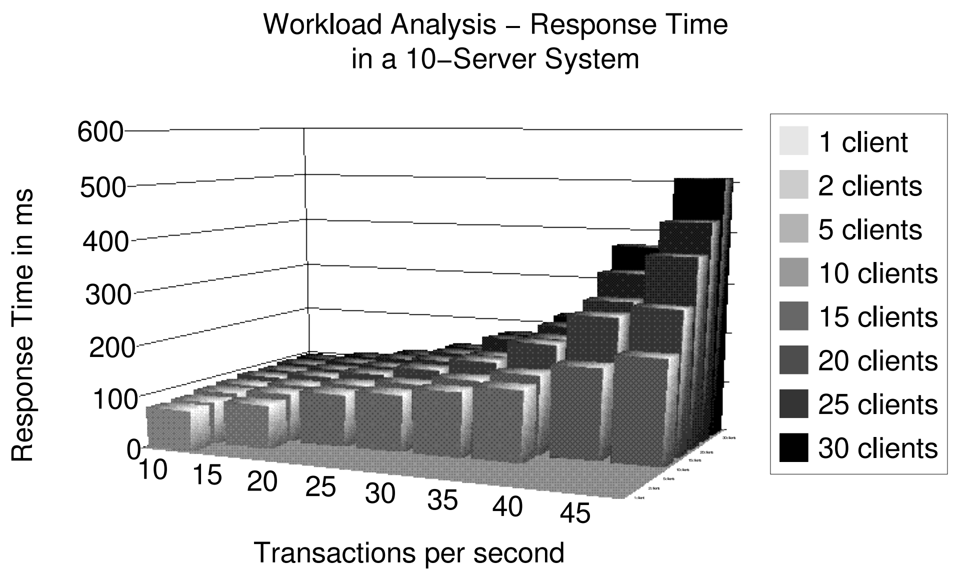

Figure 7.5: Workload I: (a) Response time of 10-node system at varying load and multiprogramming levels

around 10 ms when we move from the non-replicated system to a 2-server configuration. Further increasing the number of sites only increases this delay by a small margin. These results directly reflect the small communication overhead induced by sending the write set within transaction boundaries. Interesting and at first unexpectedly, the response times for higher multiprogramming levels decrease when more sites are in the system. We will discuss this effect in more detail in the next experiment.

As a first summary, although the figures are not directly comparable, the test demonstrates that the dangers of replication seem to have been circumvented in Postgres-R. At least for this relatively small load (10 tps), Postgres-R seems not to be significantly affected by the number of replicated nodes.

7.3 Workload Analysis I: A 10-Server Configuration

Replication is used either to improve performance or to provide fault tolerance. To improve performance, the replicated system has to be able to cope with increasing workloads and increasing number of users. Thus, the next step is to analyze the behavior of Postgres-R for high update workloads and different multiprogramming levels.

We conducted a series of tests similar to the previous experiment for a configuration of 10 servers. In this experiment, transactions consist of 10 update operations. We steadily increased the workload (throughput) from 10 to 50 tps and, for each workload, we conducted several tests varying the multiprogramming level from 1 to 30. This means, the more clients exist the less transactions each client submits to achieve a specific workload. Note that transactions have more operations and the workload is much higher than in the previous experiment. Note also that there are still only update operations in the system.

Figure 7.5 depicts the response times for this experiment. Generally, the maximum throughput is limited at small multiprogramming levels since a client can only submit one transaction at a time, and hence, the

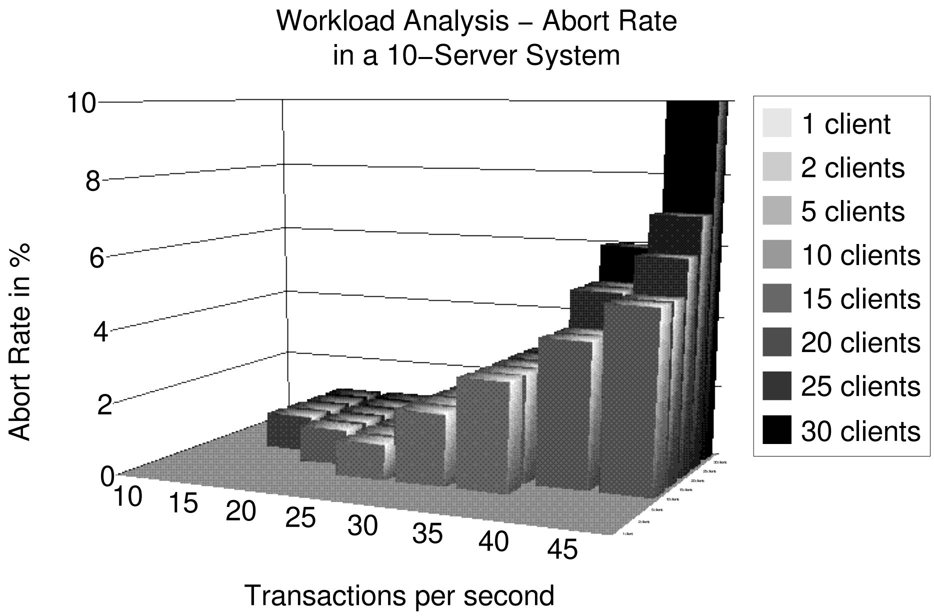

Figure 7.6: Workload I: Abort rate of a 10-node system at varying load and multiprogramming levels

submission rate per client is limited by the response time. For instance one client would have to submit a transaction each 50 ms to achieve a throughput of 20 tps but response times are close to 100 ms.

Having 10 or more clients in the system we were able to achieve a throughput of up to 45 tps before response times were longer than the required interarrival time. In general, CPU was the limiting factor. The CPU was saturated at high workloads and adding more transactions per time unit resulted in longer response times for each individual transaction. With 10 clients response time increases steadily but within acceptable ranges. Here, each site has one local client and the load is optimally distributed. Increasing the multiprogramming level beyond 10 clients results in higher response times for a given workload. The reason is mainly due to the higher administration overhead (process switches, communication to/from the client) and contention at specific resources. If there are more clients in the system there is a higher probability that two clients compete for the same resource (for instance, the latch for the log file or the lock table) even if the system workload is the same. While the impact of the additional client management is small when the workload is small (response times are similar for all multiprogramming levels), its influence increases once the CPU resource saturates (high multiprogramming levels result in very long response times at high workloads).

It is important to note, however, that the increase in response time with increasing workloads and multiprogramming levels is not due to replication. Centralized systems encounter exactly the same problems.

Figure 7.6 depicts the abort rates for this experiment. The abort rates are small over the entire test set. They are between 0% and 2% for the majority of settings and are only significant (up to 10%) for high workloads and multiprogramming levels. Here, they are a direct consequence of the higher response times. This shows how important it is to keep response times within acceptable ranges.

To summarize, the system is stable and provides good response times over a wide range of settings and workloads. The general behavior is similar to a centralized system. There is also no abrupt degradation once the CPU saturation point is reached. As a result, adding replication to the system does not need to have the performance degradation predicted by [GHOS96] even for such extreme workloads as those used in this experiment (only update operations).

7.4 Workload Analysis II: Varying the Number of Servers

After having looked at a fixed configuration, we will now compare the behavior of systems with different number of servers. As the analysis of Gray et. al [GHOS96] and the results of the first experiment show, conventional replication protocols do not cope very well with increasing system sizes. To analyze the scalability of Postgres-R, we evaluated configurations with 1, 5, 10 and 15 nodes. For each system size, we conducted the same test suites as in the previous experiment. Figures 7.7 to 7.10 show the response times with increasing throughput for 5, 10 and 15 servers. Here, each figure shows the results for a specific multiprogramming level (5, 10, 15 and 20 clients). The first two figures also include results for a non-replicated 1-node system.

Before discussing the individual figures, there are several observations that hold for all of them. First, as in the previous example, increasing the throughput increases the response time due to increasing CPU consumption. Second, in all test suites, the non-replicated 1-node system behaves significantly worse than the replicated systems. For 15 and 20 clients, response times were already over one second for 10 tps and we did not include them in the figures. In fact, the only test run in which the non-replicated system has better performance is with one client (see Figure 7.4). Third, in most cases, if we look at a specific throughput, the 15-node system has better performance than a 10-node system, which in turn has better performance than a 5-node system. Or in other words: the response times for a given throughput decrease as the number of nodes in the system increases. Furthermore, in many tests we observe that the maximum throughput increases with the number of nodes. And this, although there are still only update operations in the system that must be performed at all sites. This means, that Postgres-R is able to improve the performance and increase the processing capacity of a database by using replication even for extreme workloads.

Increased Processing Capacity In order to understand why Postgres-R can take advantage of the increased processing capacity of additional nodes even if there are only writes in the system, we need to look at how transactions are being processed. The most important fact is how remote transactions are handled. Nodes only install the changes of remote transactions without having to execute the SQL statements (see Table 6.1). As a result, remote transactions use significantly less CPU than local ones and cause generally less overhead. In addition, since updates on remote sites are fast and applied in a single step, the corresponding locks are held for a very short time. This helps to lower the conflict rates. In contrast, local transactions have more overhead because they execute the entire statement and keep locks longer, but also because they require complex client management. When there are more nodes in the system, each node has less clients submitting local transactions and more fast remote transactions which allows to run additional transactions or reduce overall response times.

Low Coordination Overhead Another important point to emphasize is that replication seems not to cause much overhead in this experiment. That is, there is no appreciable impact of the coordination overhead on the overall performance. This is the case because only one message exchange is necessary within the transaction boundaries. Furthermore, the local node does not need to wait until the remote sites have executed the transaction but only waits until the write set is delivered. In fact, we observed that the communication

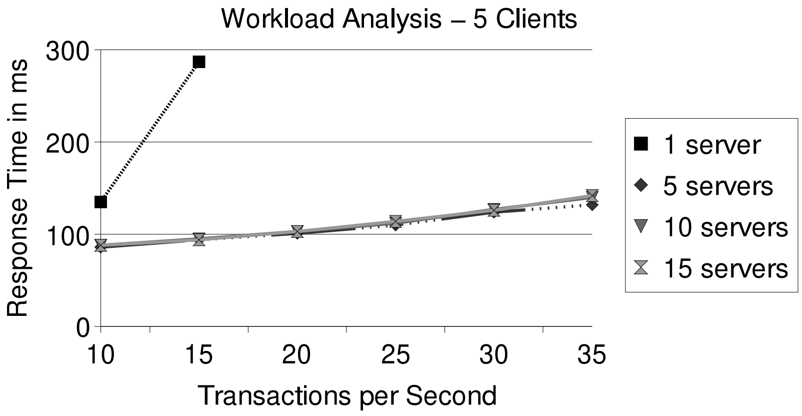

Figure 7.7: Workload II: Response time with 5 clients at varying load and system size

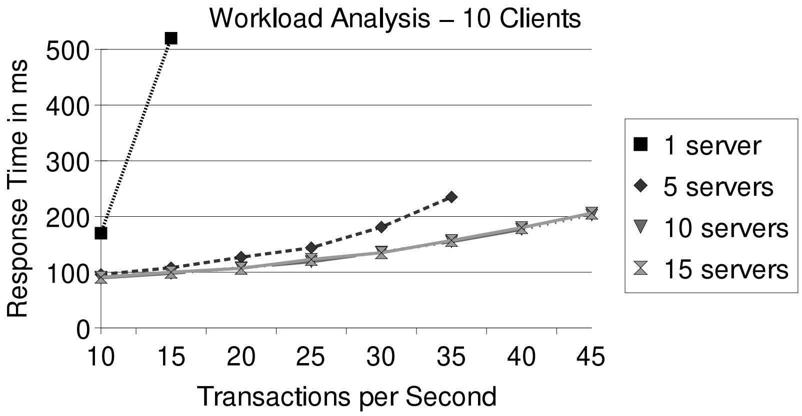

Figure 7.8: Workload II: Response time with 10 clients at varying load and system size

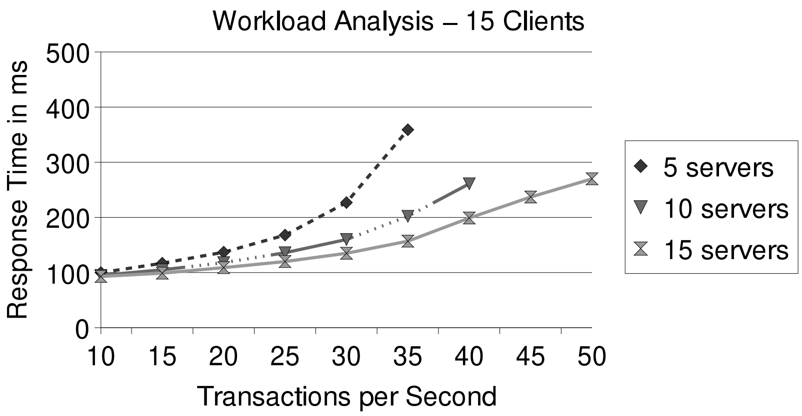

Figure 7.9: Workload II: Response time with 15 clients at varying load and system size

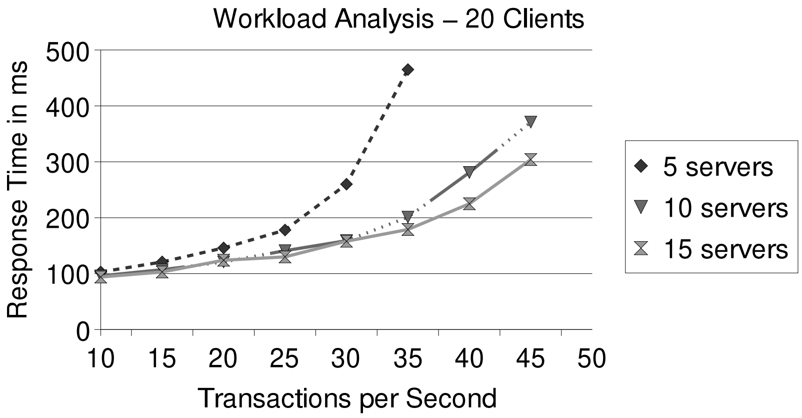

Figure 7.10: Workload II: Response time with 20 clients at varying load and number of servers

overhead was generally small. Throughout the whole experiment message delays were always between 5 and 10 ms while constructing the write set added only a few milliseconds.

Impact of the Multiprogramming Level It is, however, important to point out that the number of clients has a considerable impact on the performance differences of the different configurations. With 5 clients (Figure 7.7) the replicated system is better than the non-replicated system. While the non-replicated system has to handle 5 clients with local transactions, the 5-node system has only one local client per server and the ratio of local to remote transactions is 1:4. However, there are no performance differences for different system sizes. The reason is that 5 clients are already perfectly distributed on a 5-node system and adding new nodes cannot improve the performance. With 10 clients (Figure 7.8), we can observe that the we have a performance improvement between the 5 and 10-node system because now the 5-node system has two handle 2 clients per site. With 15 clients (Figure 7.9) the performance differences are now visible for all three system sizes. The 5-node system has to handle 3 clients per site leading to considerable contention. In the 10-client system, half of the nodes has two clients while we have perfect distribution at the 15-site system. With 20 clients (Figure 7.10), the performance improvements of the 15-site system compared to the 10-node system are smaller, especially for smaller throughputs, because now some of the sites have also 2 clients.

7.5 Workload Analysis III: Differently loaded Nodes

In a real setting, sites will have different processing capacity or will be differently loaded. As an example, in the previous experiments there were a couple of test suites in which some nodes had to handle more local clients than other nodes. Such configurations are badly handled in conventional replication protocols. Since traditional approaches use an atomic commit protocol and only commit a transaction when it is executed at all sites, slower nodes will also slow down the local transactions of fast nodes. Ideally, however, the response time would only depend on the capacity of the local node, i.e., local transactions of fast nodes have short response times, transactions of slower or heavy loaded nodes have, accordingly, longer response times.

In order to evaluate the behavior of Postgres-R in such configurations, Figure 7.11 takes a closer look at

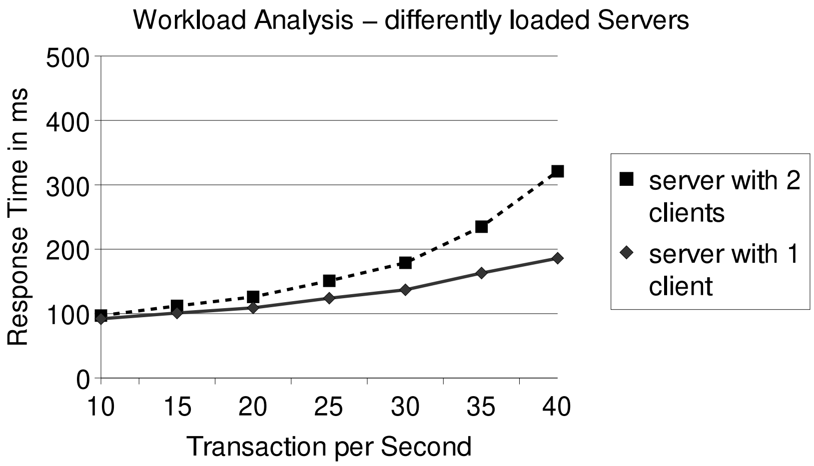

Figure 7.11: Workload III: Response time at differently loaded nodes in a 10-node system with 15 clients

a 10-server configuration with 15 clients. In this configuration, nodes are differently loaded since some nodes have one client and some have two clients. The figure presents separate curves for the response times experienced at these two types of nodes.

We can observe that when the workload increases the response time on the less loaded nodes is significantly better than on the nodes with two clients due to the lower CPU utilization. This means that slower nodes do not have a great impact on the transactions of the less loaded, faster nodes. Postgres-R achieves this by separating the local execution from the remote execution. The only coordination between the sites during the execution time of a transaction is the delivery of the write set message. Although message delay can be slightly increased on all sites if there exist slow sites in the system the impact of one message delay is small compared to the entire execution time of the transaction.

To summarize, Postgres-R is able to cope with differently loaded nodes or nodes with different processing capacity. This means, we can build a system with heterogeneous hardware or distribute the load unevenly among the sites and will still be able to provide optimal response times only depending on the capacity of the local node.

7.6 Conflict Rate

Usually, database access is not equally distributed among all data but there exist hot-spot areas that are accessed by most transactions leading to high conflict rates. As discussed by Gray et. al [GHOS96], high conflict rates are very problematic in replicated systems where traditional approaches suffer from one data item deadlocks.

In the previous experiments, conflict rates, and hence abort rates, were rather small because we modeled a uniform data access distribution and there was a low probability for two transactions to access the same tuple at the same time even for high throughputs. Abort rates only increased when response times degraded due to CPU saturation. To stress test the system we have run a suite of tests with transactions having different access distribution patterns. The database still consists of 10 tables with 1000 tuples each, but each table has a hot-spot area. The access distribution is determined by two parameters. The *hot-spot access frequency* determines the percentage of operations that access the hot-spot area. Within the hot-spot and the non hot-spot area the access has again uniform distribution. The *hot-spot data size* determines the percentage of tuples in each table that belong to the hot-spot area. For instance, a hot-spot data size of 20% means that in each table 200 tuples are hot-spot. In general, the higher the access frequency and the smaller the data area, the higher is the conflict rate. Table 7.2 depicts the tested configurations. The first data configuration (50/50) describes a uniformly distributed access without a hot-spot area. For each configuration, we have run tests with increasing workloads. The tests were performed with a 10-server configuration and 20 concurrent clients. A transaction consists of 10 update operations.

Figure 7.12 depicts the response times and Figure 7.13 the abort rates for the test runs. For clarity of the figure, the abscissa depicts the different data access configurations, while the z-axis depicts the results for increasing workloads. The leftmost column-suite (no hot-spot area, increasing workload) repeats the results that are also shown in Figure 7.5.

We can observe that the response time for a given throughput only increases slightly with increasing conflict rates. The biggest increase in response time can be observed between the 90/10 and the 90/5 configurations. The performance loss is generally due to longer lock waiting times for tuple and index locks. At higher conflict rates and workloads the abort rates and the required restarts are also an influential factor.

Looking at the abort rates, we can observe that they stay stable and only increase slightly for a wide range of configurations. The first exception is the 90/5 data access configuration which has rather high abort rates for all throughputs (already 9% for a throughput of 10 tps). The second exception are the combination of high throughput and high conflict rate. Here, abort rates degrade. If we consider abort rates of more than 10% as degradation, we can extract the following numbers from Figure 7.13: a 90/5 configuration only allows throughputs of less than 20 tps, 90/10 and 80/10 allow less than 30 tps, and 80/20 allows less than 40 tps. These high abort rates are due to two reasons. First, the implemented SER protocol using shadow copies aborts a local transaction T_i whenever a remote transaction T_j whose write set is delivered before T_i's has a conflicting operation. Since local executions are optimistically such aborts cannot be avoided for RIW/write conflicts. The second reason is the unnecessary aborts that were already discussed in Section 5.3.1. They occur when remote transactions first abort local transactions but must later also be aborted. This is possible when the write set arrives before the abort message (what will usually be the case). Note that the protocols using one message (snapshot isolation) do not have this disadvantage. Although the abort rates have an influence on the response time it is surprisingly small. The reason is that an abort takes place on average after half of the transaction has been executed, i.e. in the middle of the read phase. The abort itself is very fast (nothing has to be undone) compared to the commit. This means that if an aborted transaction can be automatically restarted by the system (what might be the case for stored procedures), the system is able to work even at high abort rates.

A side effect of the high abort rates that cannot be seen in the figure is that within a single test run (specific data configuration and throughput) response times did not stabilize. The response times shown are after the execution of 6000 transactions, but they were significantly smaller after 5000 transactions and significantly higher after 7000 transactions. While this problem also occurred in some of the other experiments (see the discussion at the beginning of this chapter) it was much more visible in this experiment at high abort rates. The reason is that at an abort rate of 50% there are not 60.000 operations executed but an additional 15.000 (calculated from: half of the transactions abort after the execution of half their operations). The additional entries in tables and indices are mainly in the small hot-spot area and result in higher and higher access times.

Generally, we can say that Postgres-R has good response times but high abort rates when conflict rates

Data configuration	I	II	III	IV	V	VI
Hot-spot access frequency in %	50	60	80	80	90	90
Hot-spot data size in %	50	20	20	10	10	5
Hot-spot data size in total # tuples	–	2000	2000	1000	1000	500

Table 7.2: Conflict Rate: Data access distribution to achieve different conflict rates

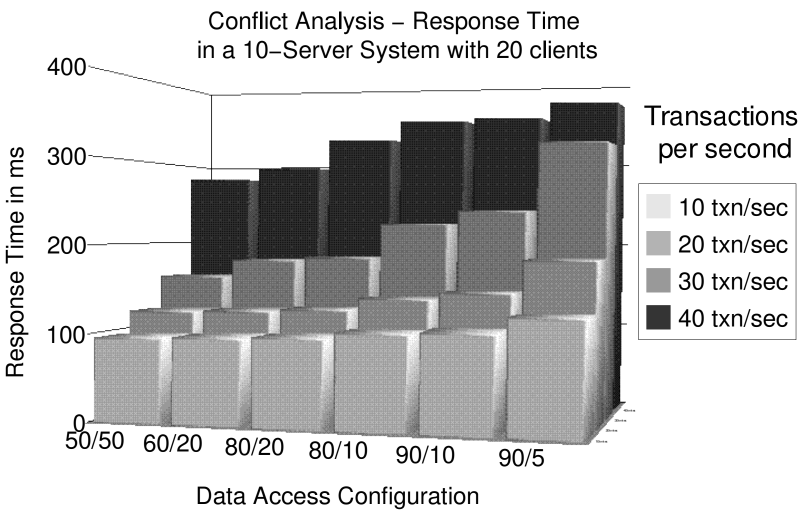

Figure 7.12: Conflict Rate: Response time in a 10-server system for varying conflict rates

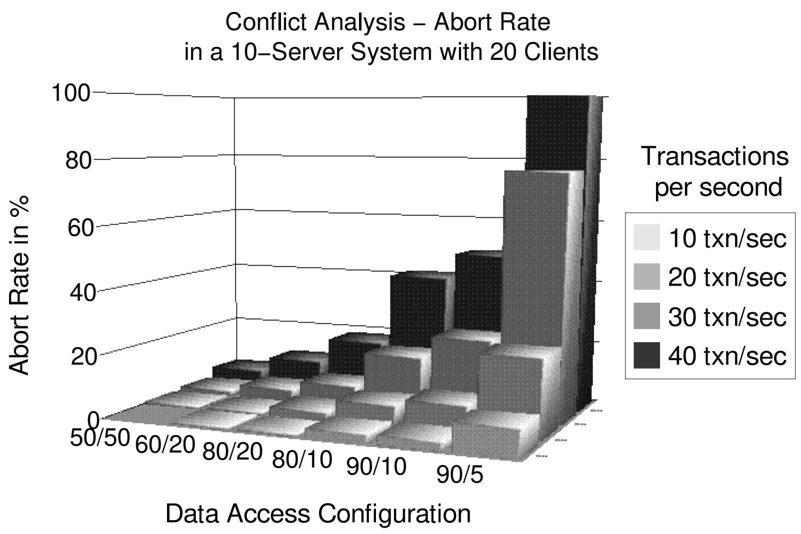

Figure 7.13: Conflict Rate: Abort rate in a 10-server system for varying conflict rates

are high. Abort rate degradation occurs when the hot-sport area contains 1000 tuples or less – which we consider a very small data size compared to nowadays Gigabyte databases. Furthermore, some of the aborts can be avoided by using algorithms that only require one message per transaction. Also the use of deferred writes can alleviate the abort problem.

7.7 Communication Overhead

One of the problems of using group communication systems is the poor performance that many of them exhibit. The claim that Postgres-R can tolerate more than 15 replicated nodes is conditional to proving that the communication system used actually scales up. In this third experiment, we analyze the capacity of the communication system to handle high message rates. Earlier performance studies have shown that the limiting factor tends to be the processing power and not the network. This is due to the high complexity of the algorithms used to derive a total order and implement delivery guarantees [BC94, FvR95a, MMSA+96]. The goal is to test whether high transaction loads can collapse the communication system and whether communication delays due to load can severely affect response times.

In the previous experiments, the number of messages never exceeded 100 messages per second (at a throughput of 50 tps with 15 nodes). Up to then, the communication system is not the bottleneck. In order to stress test the system, we performed an experiment with very many, very short transactions. These transactions consist of only one operation, thus, the write set is small but the communication overhead has a bigger impact on the overall response time. For the experiment, we used 20 concurrent clients generating a throughput between 40 and 200 tps. To have a reference, we have included the response time for 40 tps in a non replicated system.

Figure 7.14 shows the response times and Figure 7.15 the message delay for this configuration. Clearly, as the number of messages in the system increases, the communication system becomes slower. The average delay for each message goes from 5 ms to almost 60 ms. Transaction response times vary proportionally to this delay as the similar slopes in the figures indicate. A resource analysis has shown that the communication process requires most of the CPU at high transaction loads. Thus, the message delay is due to increasing message processing requirements (for message buffering, determining the total order etc.) and not to a shortage of network bandwidth. Faster processors or, even better, a multiprocessor machine would help. Observe, however, that the number of nodes has not a direct impact on the message delay. It is only the submission rate that has an effect. Thus, the claim that to support a given load, eager replication can be used to improve performance is still valid.

That this is the case can be better seen by looking at the non replicated case. While the non-replicated system has still smaller response times at 40 tps, it cannot cope with 20 clients and a workload of 100 tps due to log and data contention and the overhead of managing 20 clients at a single site. The response time degrades and is not shown in the figure. Replicating the system with 5 nodes allows to cope with this load. If 10 or 15 nodes are used the response times improve (as in the previous experiment). Similarly, 5 nodes cannot cope with a load of 200 tps. 10 and 15 nodes can. As the load increases, the improvement in response time due to replication when compared with configurations with less replicated nodes is clear.

We believe, however, that for such extreme transaction types, scalability is limited. The figures show that there are basically no performance differences between the 10 and 15-node configuration. This is partly due to the client distribution (20 clients can be perfectly distributed among 10 sites but not among 15 sites). Another reason is, however, that for these transaction types the overhead at remote sites is not much smaller than at the local site. The message overhead still occurs at all sites (the local node packs and sends the

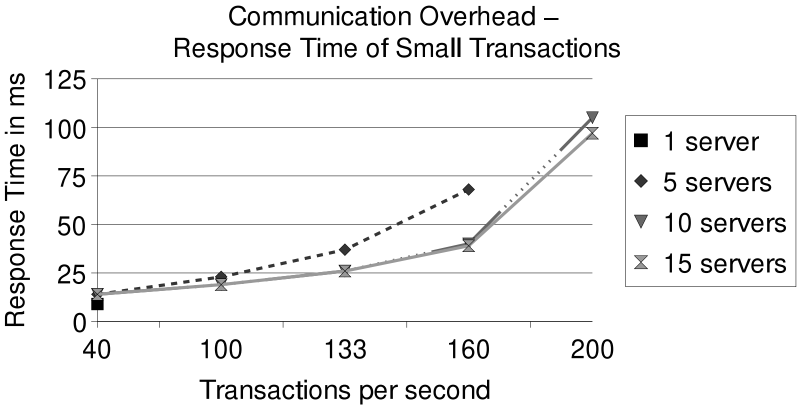

Figure 7.14: Communication: Response time at varying load and system size

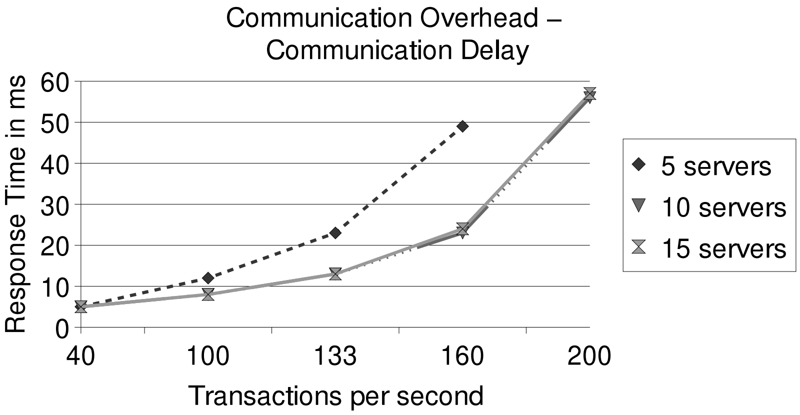

Figure 7.15: Communication: Communication delay at varying load and system size

message, the total order is determined by some coordination among all sites, and the remote sites receive and unpack the message). Also the transactional overhead (start and commit of transactions and maintenance of transactional information) occurs at all sites. Only the operation itself is faster at the remote sites. Since there is only one operation per transaction, the performance gain is not as high as with transactions consisting of many operations.

As a summary of this experiment, communication overhead is a factor to take into consideration but it only started to play a limiting role under high update transaction loads (over 150 tps).

7.8 Queries and Scalability

The previous experiments have tested the limits of Postgres-R by using extreme workloads, and so far we can conclude that Postgres-R seems to have solved most of the problems associated with eager replication. This experiment now looks at more realistic settings, analyzing the behavior of Postgres-R at workloads for which replication seems especially attractive. This is the case for applications with high rates of queries (read-only transactions).

There are two main issues to consider. The first is scalability, i.e., by adding new nodes to the system we want to increase the maximum throughput. Ideally, doubling the number of nodes should achieve double throughput. We have already shown in the previous experiments that Postgres-R scales to a certain degree even when the workload consists of only update transactions. With queries, scalability should even be better.

The second issue is how update transactions and queries interfere with each other. This problem has been discussed in detail in Section 4.1 and evaluated in the simulation study in Chapter 5. It has shown that the standard SER protocols are not suitable since they abort queries too often. In a real setting, queries are even more problematic since they usually access entire tables and hence, require relation level locks. In Section 4.1 we have proposed several alternatives to strict 2-phase locking, e.g., short read locks or snapshots. In Postgres-R, we use a simple form of the cursor stability approach (see Section 4.1.2). We acquire short relation level locks that are released directly after the operation. With this, queries are not aborted upon conflicting write operations but the updating transaction has to wait. Since the queries we use in our test runs consist of only one SQL statement on a single table they acquire only one relation level lock and hence, we still achieve serializability.

The test suites that have been run for this experiment try to imitate a cluster database where more and more nodes are added to handle increasing number of user requests. We start with one server and three clients and continuously increase the number of servers and clients using the same ratio (3 clients per server). One of the clients submits an update transaction each second (consisting of 10 operations), the two other submit queries each 150 ms (which scan an entire table). Thus, the load is around 15 local transactions per second and per node with a 14 to 1 rate between queries and update transactions. We have chosen this workload since a non-replicated 1-server system is considerably loaded but not overloaded with 15 tps (ca. 60% CPU utilization). The database size does not increase with the number of nodes. That is, as we increase the load in the system, we also increase the conflict rate.

The response times for update transactions and queries for this scalability experiment are shown in Figure 7.16. As pointed out above, by considering queries, we are able to achieve much higher throughputs up to 225 tps in a 15-node system, i.e., we have an optimal scaleup for the entire test range. The response times increase with the number of nodes but are reasonable if we take into account that the absolute number of update transactions (that must be applied everywhere and create conflicts), increases constantly.

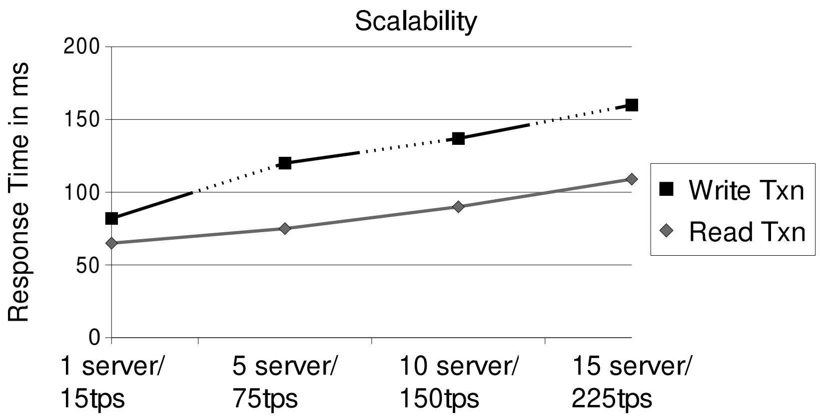

Figure 7.16: Queries and Scalability: Response Time for update transactions and queries for increasing number of nodes and throughput

In fact, conflicts start to become a problem at higher loads. Although queries are not aborted, queries and update transactions delay each other. This can be resolved by using the hybrid protocol that combines serializability for update transactions and provides a snapshot for queries. In that way, updating transactions never conflict with queries and are not delayed by them. Such a hybrid protocol practically eliminates conflicts at most loads and will allow to scale Postgres-R even further.

7.9 Discussion

The experiments of this chapter have given numerous insights into the performance of Postgres-R:

- Postgres-R does not show the degradation characteristics from which standard distributed locking schemes suffer when the number of nodes in the system increases. Instead the system is stable and provides good response times.
- Even for write-only workloads, Postgres-R is already able to exploit the increased processing power of a replicated system. Increasing the number of nodes decreases the response time for a given workload and increases the maximum throughput. This is possible for mainly three reasons. First, Postgres-R executes transactions only locally at one site. Remote sites only apply the changes. Furthermore, the use of an efficient group communication system keeps the communication overhead very small. As a third reason, distributing clients on several servers significantly reduces contention on each of the sites and hence, allows higher multiprogramming levels.
- Differently loaded nodes resp. nodes with different processing capacity, do not influence each other. That is, transactions executing on fast nodes are not delayed by slow nodes in the system. This is possible because Postgres-R decouples the local execution of a transaction from applying its updates at remote sites.
- As mentioned in Chapter 5, the SER algorithm implemented in Postgres-R suffers from high abort rates if the conflict rates are high. These aborts, however, occur only at multiprogramming levels and workloads which the centralized database system is not able to handle at all.
- The efficiency of the communication system plays a crucial role and can be the limiting factor. Existing

group communication systems, however, are able to cope with a considerable amount of messages that will be sufficient for a wide range of database applications.

- As predicted in Chapter 5, queries require a special handling of conflicts. If this is implemented, Postgres-R is able to provide scalability and the tested configuration of 15 nodes is able to process far over 200 transactions per second. We believe that even a higher throughput can be achieved if queries use snapshots and do not acquire any locks.

The results show that eager update everywhere replication can be used in cluster configurations to provide consistent and efficient data maintenance for a variety of workloads. This means, the main goal of cluster computing – increasing throughput by adding new nodes to the system and distributing the load across the system – can be supported by eager replication without serious limitations on response times. We are confident that other database systems implementing this approach will achieve the same or even better results.

8 Recovery

So far, we have presented a framework for efficient eager replica control and proven its feasibility by integrating it into an existing database system. This chapter evaluates a further important topic that needs special treatment: recovery.

In chapter 4, we have shown that the replicated database system is able to handle site or network failures by taking advantage of virtual synchrony. The communication system informs the database system about failures in form of view change messages excluding unreachable sites. This happens in such a way that each site can individually decide on the outcome of pending transactions avoiding any extra communication.

In the same way as failed nodes must be excluded, the replicated system must be able to recover previously failed sites and to include new sites into the system. This, however, is not as straightforward as the exclusion of nodes. Before a new node can start executing transactions, its database must be identical to the databases in the rest of the system. Two key issues must be handled. First, the current state of the database has to be installed efficiently at the joining node, interfering as little as possible with concurrent processing of transactions in the rest of the system. Second, a synchronization point must be determined from which on the new node can start processing transactions itself. This chapter explores several solutions for node recovery. We will present the different steps involved in the recovery process and for each of the steps discuss optimizations and implementation alternatives. Note that this chapter provides solutions that are, in general, not restricted to a specific database system. However, some of the recovery steps depend on the specifics of the underlying database system and we point out when this is the case. The last section of this chapter describes how we have implemented recovery in Postgres-R.

8.1 Basic Recovery Steps

The process of joining a node can be divided into different steps:

- **Single Site Recovery** A recovering node first needs to bring its own database into a consistent state. This can be done using standard undo/redo recovery [BHG87, MHL$^+$92] using its own log. A new node does not need to perform this step.
- **Reconciliation** Care has to be taken if the system only works with reliable message delivery. In this case, as pointed out in Section 4.3, a recovering node might have committed transactions before its failure that are not visible to the rest of the system, or might have committed transactions in different order than the available sites. These updates must be reconciled, i.e., they must be undone in the recovering node.
- **Data Transfer** A peer node (one of the available nodes in the system) must provide the joining node with the current state of the database. The simplest solution is to send an entire copy (this is the only solution in the case of a new node). Alternatively, it can send parts of its log that reflect the updates

performed after the recovering node failed. Which option is best depends on the size of the database and the number of transactions executed during the down-time of the recovering node.

- **Synchronization** Since the system should be able to continue transaction processing during the database transfer, the installation of the database must be synchronized in such a way that for each transaction in the system, either the updates of the transaction are already reflected in the data transfered or the joining node is able to process the transaction after the database transfer has successfully terminated.

Terminology Before we discuss these points in more detail, we will introduce some terminology that is used throughout this chapter. N_n depicts a new node, N_r depicts a recovering node and N_j depicts any type of node joining the system (new or recovering). The peer node transferring the current database state to the joining node is denoted N_p. Each transaction T has a globally unique identifier $gid(T)$ and the transaction is also denoted as T_{gid}. This global identifier is the sequence number of its write set, i.e., the position of T's write set in the total order of all delivered write sets. Note that the global identifier $gid(T_i)$ for transaction T_i is the same as the EOT timestamp $TS_i(EOT)$ in the Snapshot Isolation (SI) protocol introduced in Section 4.1. Since the gid is generated rather late in the life time of a transaction (only after the delivery of the write set), we assume that T also has some local transaction identifier $lid_N(T)$ at each node N usually generated at T's start time and T is referred to as T^N_{lid}. Using the sequence number as a global identifier instead of a combination of host and local transaction identifier is useful, since the sequence numbers represent the commit order of the transactions.

A central component of the recovery procedure is the log file. Each node N maintains its own log, $L(N)$. It is needed for the standard single site recovery but we will, whenever possible, also use it to support some of the other recovery steps. During normal transaction processing different types of log entries are created and appended to the log. The different entries of the log will be described step by step in the following sections.

8.2 Single Site Recovery

Within this section we present the basics of any type of single site recovery [BHG87]. We would like to point out that many different types of logging and recovery are implemented in database systems and the recovery process of our replicated system has to somehow adjust to the given single site recovery components.

In here, we assume that when a site fails, its processors stop working and the main memory can be left in an undefined state while stable storage is not affected (media recovery, for instance, is discussed in [BHG87]). When the failed site restarts, parts of the valid database might have been lost and the single site recovery must lead the database back to a consistent state. Recovery is needed because most systems use a *noforce/steal* buffer management strategy during normal processing, i.e., modified data is not forced to disk at commit time (noforce) but it is possible that updates are flushed to disk before the transaction terminates (steal). Therefore, recovery consists of two parts. First, the updates of aborted transactions that are still reflected in the database must be undone. Also all transactions that were active at the failed node at the moment of the failure must be aborted. This part is called *undo recovery*. Second, updates of committed transactions that were not propagated to disk before the failure must be redone (*redo recovery*).

In order to perform recovery, undo and redo information must be written to the log during normal processing. To be efficient, writing the log must be much faster than writing to the database itself. Therefore, the log is a sequential file where the log-tail resides in main memory and is flushed to a dedicated log disk at certain points in time. To be able to perform undo recovery, the corresponding log information must be flushed to

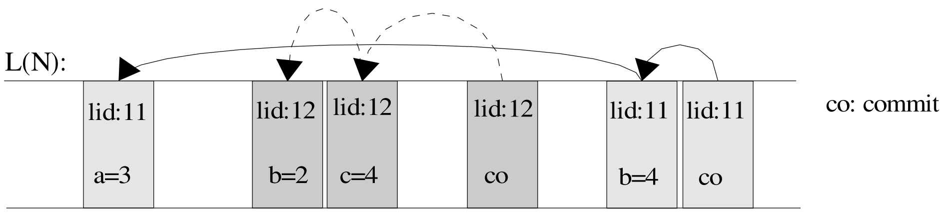

Figure 8.1: Backchained log entries of two transaction in the log of node N

disk before the modified data is written to the database, for redo recovery the log must be flushed before the transaction commits. There exist basically two types of log entries in the log:

- For each update of transaction T on data item X a *write-log entry*, containing at least T's identifier, X's identifier, the before-image of X to perform an undo operation, and the after-image of X to perform a redo operation.
- For each transaction T an *EOT-log entry*, containing T's identifier and the termination decision (commit or abort).

In the context of single site recovery, $lid_N(T)$ can be used as the transaction identifier. The log entries of a single transaction are usually chained (for fast forward and backward search). Additional information, like, e.g., log sequence numbers in the log and on database pages are used to optimize the recovery process. As an example, Figure 8.1 depicts the log entries of two transactions T_{11}^N and T_{12}^N.

The undo and redo recovery steps depend of the type of before- and after-images and the granularity of a data item. A data item can be an entire physical page, a part of a page or a logical unit like a record (tuple) of a table. The before- and after-images can log physical information, i.e., they represent the physical content of the entire data item or of those parts that have been modified. In this case, redo and undo are simple copy operations. Alternatively, they can be logical and consist of the description of the operation. Then, a redo has to redo the operation, an undo performs the inverse operation.

Recovery is usually performed in different passes. An *analysis pass* checks which transactions have committed, aborted or were active at the time of the failure (active transactions must be aborted). An *undo pass* usually scans the log backwards and undoes all updates of aborted transactions that are reflected in the database. The *redo pass* scans the log in the forward direction and redoes all operations of committed transactions that have not been reflected in the database. The order of the different passes and the exact operations depend strongly on the log information. Usually, checkpoints are performed periodically during normal processing limiting the log space that has to be parsed for recovery purposes. Detailed discussions of single site recovery can be found in [BHG87, GR93, MHL$^+$92].

The single node recovery is the first step in the recovery of a failed node in a replicated system. It can be done completely independent from the other sites and does not interfere at all with concurrent transaction processing in the rest of the system.

8.3 Reconciliation

As discussed in Section 4.3, using reliable message delivery, a failed node might have wrongly committed transactions. These transactions must be detected and reconciled. In the same way, if uniform reliable message delivery is used for the write set but reliable delivery for the commit message and blocked transactions are aborted, reconciliation is also necessary. Note that if uniform reliable delivery is used for all messages, reconciliation is not needed, since failed nodes will never have wrongly committed transactions.

In order to perform reconciliation, N_r has to compare its log $L(N_r)$ with the corresponding parts of $L(N_p)$ of a peer node. Let V_i be the view during which N_r failed, that is, N_r was member of V_i but excluded in view V_{i+1}. Furthermore, let N_p be a peer node which was available during V_i, i.e., that also installed V_{i+1}. For all transactions for which all messages were delivered before view change V_i it is assured that N_r has decided the same outcome than N_p (see Theorems 4.3.1 and 4.3.3). Potentially wrongly committed transactions are all those for which at least one message was sent in V_i.

8.3.1 Log Information needed for Reconciliation

In order to detect all possible mismatches we must extend the single site log to contain additional information:

- At least the EOT-log entry of a transaction T must also include the global identifier $gid(T)$. If the log entries of a transaction are linked or if the termination entry contains both the global and the local identifier, the log entries belonging to a transaction can easily be determined in a backward pass.
- Whenever a view change message is delivered indicating a new view V_i, a view change entry is included in the log. In the SI protocol this entry (indicating the start of V_i) must precede all EOT-log entries of transactions whose write sets are delivered in V_i (i.e., after the view change message). In the SER/CS/Hybrid protocols it must precede all EOT-log entries of transactions whose decision message is delivered in V_i (i.e., the write set could have been delivered in an earlier view).

Note that we do not impose restriction of when the write-log entries are written to the log. They can be written at the time the write set is delivered (this will be the case for remote sites) but also any time earlier. For instance, if the local site performs updates during the read phase the log entries can already be written to the log during the read phase before the write set is even sent. This provides a maximum of flexibility in regard to how transactions are generally executed.

8.3.2 Determining the Transactions to be Reconciled

Determining the transactions to be reconciled depends on the replica control protocol:

- For the SI protocol, there is only one message per transaction. This means the only transactions to be reconciled are those for which the write set was delivered at N_r during V_i but not at the V_i-available nodes or where the delivery order was different. The view change entry provides a delimiter in the log to restrict the part of the log where the EOT-log entries of these transactions can be found.
- For the SER/CS/Hybrid protocols things are more complicated since there are two messages per transaction and the messages can be delivered in different views. For a transaction T to be reconciled N_r must have received both the write set and the commit before its failure but the V_i-available sites either did not receive the write set or ordered it differently, or they only received the write set but not the commit message (and then aborted T because it was in-doubt). In the first case both the write set and the commit message must have been delivered at N_r during V_i, in the latter case at least the commit message must have been delivered at N_r during V_i.

Example As an example for the SI protocol, Figure 8.2 depicts the logs of a node N_r failing during V_1 and an available peer node N_p. N_p has two local transactions $T_3^{N_p}$ and $T_4^{N_p}$ (i.e., $lid_{N_p} = 3$ and 4) and N_r

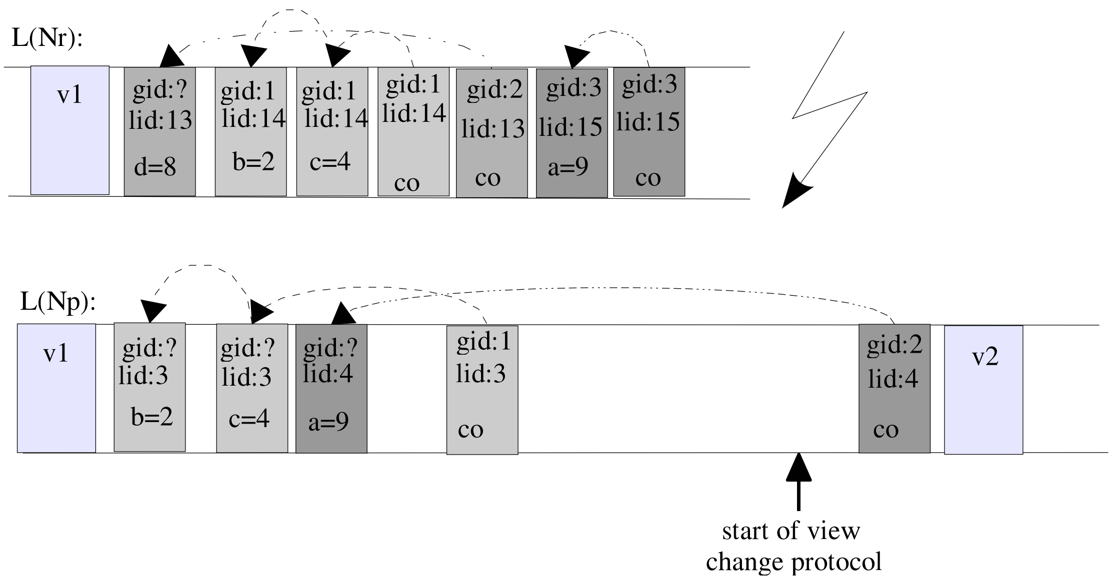

Figure 8.2: Logs at failed node N_r and available node N_p

has a local transaction $T_{13}^{N_r}$ ($lid_{N_r} = 13$). We assume that the transactions do not conflict. Both nodes send the write sets and the communication system decides on the order $T_3^{N_p} < T_{13}^{N_r} < T_4^{N_p}$. All three write sets are delivered at N_r and N_r assigns the global identifiers $gid(T_3^{N_p}) = 1, gid(T_{13}^{N_r}) = 2$ and $gid(T_4^{N_p}) = 3$ to the transactions. Furthermore, $T_3^{N_p}$ receives local identifier 14, and $T_4^{N_p}$ receives local identifier 15 for internal use. All three transactions commit and N_r writes the commit log-entries. N_p receives $T_3^{N_p}$ and assigns it $gid = 1$. However, N_p does not receive $T_{13}^{N_r}$ before N_r fails. During the view change protocol the communication system delivers $T_4^{N_p}$'s write set to N_p before it delivers view change message V_2. Hence, N_p assigns $gid = 2$ to $T_4^{N_p}$ and commits the transactions. As a result, N_r has committed a transaction that is not committed at N_p, and the nodes have assigned different global identifiers to $T_4^{N_p}$.

Reconciliation Lists and Reconciliation Logs To detect these mismatches the steps described in Figure 8.3 create *reconciliation lists* and *reconciliation logs* on N_r and N_p containing all transactions that must possibly be reconciled. By comparing the lists and logs from both sites the transactions that in fact must be reconciled can be determined. In the simple example of Figure 8.2, the reconciliation lists are $RL(N_r) = 1, 2, 3$ and $RL(N_p) = 1, 2$, the reconciliation logs are exactly the entries of Figure 8.2. When comparing both reconciliation logs N_r detects that the two transactions with $gid = 2$ are not identical and hence, aborts its own transactions with $gid = 2$ and $gid = 3$. Note that the latter transaction was in fact committed at N_p but with global identifier $gid = 2$. We still prefer to abort the transaction at this point and redo it later during the database state transfer to guarantee identical information at all sites.

Although reconciliation requires cooperation with a peer node, it does not interfere at all with concurrent transaction processing since the peer node only needs to provide parts of its logs that are on disk.

I. N_r makes a backward-pass through its log and creates a reconciliation list $RL(N_r)$ containing the global identifiers of all committed transactions for which in the case of SI the write set was delivered in V_i, in the case of SER/CS/Hybrid the commit message was delivered in V_i. Additionally it extracts a reconciliation log $RLog(N_r)$ containing the log entries of all transactions in $RL(N_r)$. When N_r reaches the view change entry for V_i it has determined $RL(N_r)$. It only needs to scan further backwards if some write-log entries for these transactions are ordered before the view change entry. Let $gidmin$ be the smallest gid and $gidmax$ the biggest gid of transactions in $RL(N_r)$.

II. N_r requests the corresponding reconciliation list and log from its peer node. N_p scans its log starting with view change entry V_i until all EOT-log entries of transactions with gid between $gidmin$ and $gidmax$ are found. Since the log entries of a transaction are back-chained all entries can be found easily from there.

III. N_r compares transactions with the same global identifiers from both logs starting with $gidmin$ and continuing in ascending order. As long as the two transactions are identical (we assume that this can be determined by comparing the write-log entries of the transactions) and are committed at both sites, the transaction is removed from $RL(N_r)$ and the next two transactions are compared. Otherwise the comparison stops and N_r reconciles, i.e., aborts, all transactions that are still listed in $RL(N_r)$.

Figure 8.3: Determining the transactions to be reconciled

8.4 Synchronization

The joining node N_j and the peer node N_p providing the current state of the database have to coordinate the database transfer in such a way that N_j does not miss the updates of any transaction. For this purpose, virtual synchrony provides are very simple synchronization point. When a node wants to join the group (after single site recovery and reconciliation) it submits a "group join request" which is a specific primitive provided by the group communication system. As with node failures, the (re-)joining of N_j to the group provokes a view change from a view V_{i-1} to a consecutive view V_i. While the failure of a node leads to a shrinking of the view (i.e., the failed node $N \in V_{i-1}$ but $N \notin V_i$), a join results in the expansion of the view (i.e., joining node $N_j \notin V_{i-1}$ but $N_j \in V_i$). The view change message V_i is delivered both at N_j and N_p (it is the first message delivered at N_j) and presents the logical time point from which on N_j is aware of all transactions, since it receives all messages sent after the view change. Hence, the view change provides us with a natural synchronization point.

Let V_i be the view in which N_j joins the system. The peer node N_p has to provide to N_j the database state including the updates of all transactions for which the write sets were delivered before V_i. Once N_p has applied these updates it will first execute the transactions that is has already received in the new view V_i and then start its own transactions.

In the case of protocols with two messages (SER/CS/Hybrid), care has to be taken when the two messages are sent in different views. It is possible that the write set of a transaction T is sent and received in view V_{i-1} while the commit/abort message is sent in a later view. Since N_j does not receive the write set, the transfered data has to contain T's updates if T commits. Therefore, N_p has to wait for the decision message to be able to apply the updates (since the decision message has no order properties, the delivery of this message does not interfere with concurrent write set messages). Additionally one has to be aware that N_j receives the decision message without ever receiving the write set. Since N_j does not know that the write set was delivered in the previous view, N_p must inform N_j accordingly.

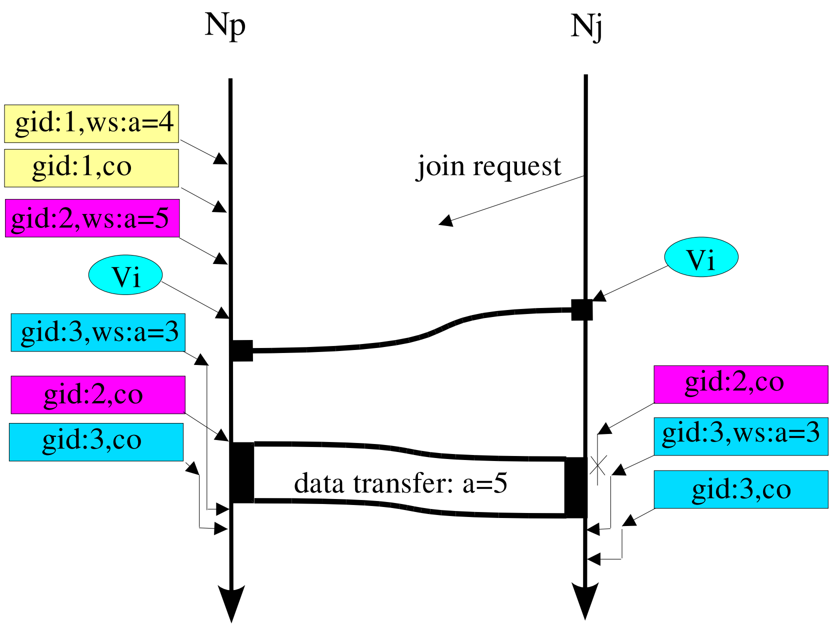

Figure 8.4: Synchronization between joining node N_j and peer node N_p

Figure 8.4 shows an example where the view change message V_i including node N_j is delivered after the write sets of transactions T_1 and T_2 but before the write set of T_3. Peer node N_p has to first wait until it receives the commit message for T_2 and then transfers the database state to N_r before applying the updates of T_3. N_r delays the incoming messages until it receives the database state. It must ignore the commit message for T_2 since its updates are already reflected in the transfered database. Once the database is installed, N_j applies the delayed transactions and from then on can start executing its own transactions.

8.5 Database Transfer

This section describes what has to be done to transfer the database state from the peer node N_p to the joining node N_j. We first concentrate on transferring an entire database copy and then discuss in the next section how the log can be used and what types of optimizations are feasible. Transferring the entire database is necessary in some cases. If the joining node is new and has no copy of the database, N_p must transfer the entire database in any case. But also if a node recovers there are situations where it might be the preferable option. This is, for instance, the case when N_r was down for a long time and during this down-time the system has processed thousands of transactions updating most of the data in the database. Determining and extracting the updated data might be more cost- and time-intensive than a simple database copy.

The basic idea is to treat the data transfer as its own independent data transfer transaction DT executed at the peer node N_p. DT reads and transfers the entire database and, in order to provide the synchronization point described above, it is serialized in regard to update transactions according to the logical time point of the view change V_i. This means all update transactions whose write sets have been delivered before view V_i are serialized before DT, all transactions who are delivered in V_i or later are ordered after DT.

Using the SER/CS protocol, the recovery manager of the peer node N_p controls the data transfer in the following manner:

I. *Synchronization:* Upon delivery of V_i including N_j, N_p creates a data transfer transaction DT which requests in an atomic step a set of special read locks. This set must cover the entire database. For each read lock:
   1. Order the read lock after all write locks from transactions whose write sets were delivered before view change message V_i.
   2. Do not abort any transaction (local transactions in their read phase can still continue)
   3. Mark the read lock as special, i.e., successive transactions acquiring write locks do not abort DT but wait until DT releases the read lock.

II. *Data Transfer:*
   1. Whenever one of the read locks is granted the corresponding data is read and transferred to N_j. N_j installs the data and confirms this to N_p.
   2. When N_p receives this confirmation, it can release the corresponding lock and normal transaction processing can continue on this part of the data.

Figure 8.5: Database transfer using SER/CS

Using the SI/Hybrid protocol, the recovery manager of the peer node N_p controls the data transfer in the following manner:

I. *Synchronization:* Upon delivery of V_i including N_j, N_p creates a read-only data transfer transaction DT. The BOT timestamp $DT(BOT)$ is set to the EOT timestamp $TS_i(EOT) = gid(T_i)$ of the transaction whose write set was the last to be delivered before view change message V_i.
   1. For each data item X read the version of X labeled with T_j where T_j has the highest $TS_j(EOT) = gid(T_j)$ so that $TS_j(EOT) \leq DT(BOT)$.
   2. Care has to be taken that DT in fact reads the correct value of X. There could still be some transactions active that were delivered before V_i and want to update X. DT must delay the read operations until these transactions have finished (as the SER/CS protocols do by requesting read locks).

II. *Data Transfer:* For each data item X, send it as soon as it is read. Once all data is sent DT terminates.

Figure 8.6: Database transfer using SI/Hybrid

The execution of DT depends on whether the SER/CS protocols or the SI/Hybrid protocols are used and is given in Figures 8.5 and 8.6. While SER/CS uses read locks the SI/Hybrid protocols use a snapshot, since DT is a read-only transaction. We assume that upon the delivery of a view change message V_i including a joining node N_j each site which was already member of V_{i-1} checks whether it is the peer node N_p. The check must be deterministic in such a way that exactly one site chooses to be the peer node N_p.

There are two advantages of the snapshot approach. First, the data transfer does not acquire any locks and hence, does not delay any transactions at the peer node N_p. Furthermore, if N_p fails during the recovery, another node can take over the recovery. Since not necessarily the current version of the data is read but the data is reconstructed, the new peer node can continue where the previous peer node failed. In contrast, using read locks, the data is locked until it is transfered. Furthermore, the failure of the peer node requires the recovering node to leave and rejoin the group to receive a new synchronization point.

Note that transaction processing is delayed on the peer node and the joining node. The rest of the system, however, can continue unhindered.

I. Make an analysis scan through the log and set $\mathcal{T} = \{T \mid gidmax < gid(T) \leq gidlast$ and T committed $\}$
II. Set $\mathcal{D} = \{\}$ and start scanning the log $L(N_p)$ backwards.
III. As long as $\mathcal{T} \neq \{\}$ read the next log entry (from the back):
   1. If it is an write-log entry on data item X of a transaction $T \in \mathcal{T}$: if $X \notin \mathcal{D}$ then $\mathcal{D} = \mathcal{D} \cup X$ (read X from the database).
   2. If it is the first log entry of T (no link to previous entry exists) then set $\mathcal{T} = \mathcal{T} \setminus T$.
IV. As soon as a new data-item is included in $\mathcal{D}$ it can be transfered to N_r.

Figure 8.7: Transferring the updated data items using the log $L(N_p)$

8.6 Optimizing the Data Transfer

There are many cases where it is more efficient to only send the changed data and not the entire database to the recovering site N_r. If N_r has been down for very little time or if big parts of the database are mainly read and seldomly written it might be more efficient to determine and send only that part of the database that has been updated since the recovering site failed.

To do so, it has to be determined which transactions N_r actually missed. For that, N_r determines after its single site recovery and after the reconciliation the transaction T for which the following holds: T committed at N_r and has the highest global identifier $gidmax$ so that all transactions at N_r with smaller global identifier also terminated. Hence, N_p has to send the data that was updated by committed transactions with global identifiers larger than $gidmax$.

8.6.1 Filtering the Database

A very simple option exists if each data item is labeled with the transaction T that was the last to update the item. This option exists, for instance, with snapshot isolation (SI). In this case, the entire database is still scanned but only the relevant data items are sent. As an example, the protocol in Figure 8.6 can simply be enhanced as follows:

II. *Data Transfer:* For each data item X: if the reconstructed version of X is labeled with T for which $gid(T) > gidmax$ then send the data item. Otherwise ignore it.

8.6.2 Filtering the Log

If data items are not labeled or the database is huge compared to the part of the log that has been created since N_r failed, N_p has the option to scan its log in order to determine the data that has been changed. Once the data items are identified they can be read (SER/CS) from the database or reconstructed (SI,Hybrid), and transferred.

In order to determine the relevant data N_p waits until all transactions that have been delivered before the view change V_i have terminated. Let $gidlast$ be the global identifier of the last transaction delivered before V_i. N_p makes a backward pass through its log and creates a set $\mathcal{D}$ containing all data items that were updated since the recovering node failed.

Figure 8.7 describes the different steps in determining the updated data items. In order to not miss the updates of any transaction, step I creates a set $\mathcal{T}$ containing all transactions whose updates must be considered. Step III.1 of the algorithm guarantees that each data item is only sent once even if the data item was updated by more than one transaction during the down-time of N_r. This is important since during the down-time of T_r thousands of transactions might have been executed, many of them updating the same data items. It would be impossible for T_r to reexecute all of them but it should only install the final effect of these transactions.

Note that N_p must also send enough of the log information to N_r so that N_r can be peer node for another failed node later on. However, it is not necessary to send the entire missing part of the log. In fact, it is enough for each data item X in $\mathcal{D}$ to send the log entry of the last transaction T that updated X and the EOT entry for T. Furthermore, the links for these entries must be reset correctly.

8.6.3 Reconstructing the Data from the Log

Depending on the structure of the log, it might even be possible to reconstruct the data items from the log and avoid accessing the database. If this is possible N_p does not need to lock the database and transaction processing can continue unhindered even in the case of the SER and CS protocols.

Reconstruction is straightforward if a write-log entry contains the complete physical after-image of the data item. In this case step III.1 of the algorithm in Figure 8.7 does not need to access the database but simply stores the after-image in $\mathcal{D}$.

However, the after-image does not always contain the entire data item. Sometimes, only the part of the data item that was updated is logged (i.e., for instance, single attributes of a tuple). In this case reconstruction might still be possible, but is more complicated because the log entries must be merged in such a way that the data sent reflect all and if overlapping the latest updates. Also care has to be taken to integrate the synthetic after-image with the stale data item at N_r.

If the after-image describes the logical operation performed instead of the physical result all and not only the latest log entry on a data item must be sent and all operations must be replayed at N_r. This is an option that we consider, as discussed above, unfeasible.

It is also possible that the after-image does not even represent the unit of a data item but the physical page updated or simply a byte stream on a physical page which is identified by the page number and offset within the page. Also here, the after-image is of little use.

8.6.4 Maintenance of a Reconstruction Log

Any of the options described above – sending the entire database or scanning the database respectively the log – involves a considerable overhead at the time of the view change. In some of the cases the database is locked during the data transfer and transaction processing on the peer node is delayed. The solutions presented so far optimize on the overhead during normal processing, i.e., they all present solutions where basically no additional overhead during normal processing takes place. As a consequence, the data transfer during the recovery is time consuming.

If nodes fail and recover frequently it might pay off to maintain additional information during normal processing to allow for a fast data transfer in the case of a recovery. The idea is to maintain a *reconstruction log RecLog* at each site which can directly be transfered to a recovering node. *RecLog* is not a sequential log but can rather be seen as a shadow database that contains entries for those updates that have been performed since a certain point in time. These entries are similar to the log entries, that is, they contain at least the identifiers of the modified data items and the global identifiers of the updating transactions. If space is not a

The $RecLog$ at node N is maintained in a background process and consists of two parts:

I. *Adding the updates of the newest transactions periodically in commit order to $RecLog$:* For each data item X updated by a newly committed transaction: if X is already in $RecLog$, replace the old entry by the new entry, otherwise insert a new entry.

II. *Deleting entries from $RecLog$ that are no more needed:* Whenever N gets informed that $gidmax_{min}$ has changed due to a recovery or – if there are currently no failed nodes in the system – through regular exchange of the $gidmax$-values of the different sites, then set $gidmax_{min}$ to the new value and delete each entry of a transaction T for which $gid(T) < gidmax_{min}$. Note that $gidmax_{min}$ can only increase but not decrease.

Figure 8.8: Maintaining the reconstruction log $RecLog$

limiting factor they can also contain the data item itself (i.e., the physical after-images). For each data item there exists at most one entry in $RecLog$, namely only the last update on the data item.

An update on a data item should be reflected in $RecLog$ if there is a node in the system that might not have seen this update; this is the case, for instance, if the node failed before it was able to commit the corresponding transaction. As defined before, let for each site N – failed or not failed – be $gidmax_N$ the global identifier of committed transaction T so that T and all transactions with smaller global identifiers have terminated. Let $gidmax_{min}$ be the minimum of all these identifiers. If the reconstruction log of a peer node N_p contains all updates of transactions with global identifiers bigger than $gidmax_{min}$, then N_p is able to use it to easily transfer data to any recovering site.

Building and updating $RecLog$ can be done by a background process and run asynchronously during normal transaction processing whenever the system is idle. This means it does not necessarily need to reflect the latest changes of the database. Upon a view change the peer node will only need little time to bring $RecLog$ up-to-date and transfer it to the recovering node. Figure 8.8 shows how $RecLog$ at a node N is created and maintained. In order to perform step I, the background process can scan the original log or transactions can trigger the update of $RecLog$ at commit time (whereby updating $RecLog$ is executed outside the scope of the transaction). To make maintenance efficient, we suggest to implement $RecLog$ as a special table in the database where each record represents one entry. There are two indices needed for the table. The first one is an index for the global identifier. This allows for fast range queries (e.g., "give me all entries with gid bigger than..."). These are essential in order to extract the part of $RecLog$ to be transfered in case of a recovery or to be deleted after a change of $gidmax_{min}$. The second index is on the data item identifiers. When a new transaction updating data item X is added to $RecLog$, this index is needed to check whether there exists already an entry for X that must be replaced or whether a new entry is inserted. Different forms of clustering (e.g., per table or by gid) can further speed up the access in specific situations.

In order to guarantee an efficient maintenance of $RecLog$ and fast data transfer, $RecLog$ should be reasonable small. If a site is down for a very long time, forcing to keep $gidmax_{min}$ low on the available sites, $RecLog$ might grow too big. To avoid such situations and keep $RecLog$ small, the site could be excluded from the option to be recovered by $RecLog$ but would be treated as a new node at recovery time and receive the entire database.

A peer node N_p now performs the data transfer using its $RecLog$. The steps are depicted in Figure 8.9. Finding the relevant data to be transfered is now reduced to the standard database operation of selecting from a table a set of records that fulfill a certain requirement.

Upon a view change message V_i including recovering node N_r, the peer node has to perform the following steps:

I. If the $RecLog$ does not contain the after-images of the data items, then acquire read locks on the database.
II. Include the last missing transactions whose write sets have been delivered before view change message V_i into $RecLog$.
III. Select all entries from $RecLog$ being created from transactions T_{gid} so that $gid > gidmax_{N_r}$. If the entries in $RecLog$ do not contain the after-images then read the corresponding data items from the database.
IV. Transfer the data items to N_r.

N_r installs the data items in its database and also updates its own $RecLog$ so that it is able to become a peer node by itself.

Figure 8.9: Transferring the updated data items using $RecLog$

8.6.5 Discussion

Which of the data transfer solutions to choose depends on several parameters. First, for some of them specific characteristics must be given (each data item stores the identifier of the updating transaction, logs store physical after-images, etc.). But the efficiency of the solutions also depends on the size of the database, the percentage of data items updated since the recovering node failed, the cost of scan operations, the cost of maintaining a reconstruction log etc.

Another important question is how easy each solution can be integrated into an existing system. Transferring the entire database requires only little changes to the existing system (only the global transaction identifiers are needed if reconciliation is necessary) and implementing the transfer procedure is straightforward. Using the log and possibly the reconstruction log adds more complexity to the system. This, however, might pay off in many situations.

8.7 Recovery in Postgres-R

Due to the very limiting logging component of PostgreSQL (no undo/redo logs are written), the current version of Postgres-R transfers the entire database to a joining node [Bac99]. The database transfer is supported rather well by PostgreSQL since it provides a function that extracts the entire database in form of SQL statements and stores it in an external file. The schema description of the tables is given in form of `create table` statements. Similar statements exist for indices, triggers, etc. The records themselves are described as `insert` statements. This is exactly the format that is needed. We cannot simply transfer the physical UNIX files in which the tables are stored since they contain many physical information that is specific to the storage system at the peer node. For instance, all objects in Postgres-R, including tables, have object identifiers. These identifiers are generated individually at each site and cannot be transferred to the joining node. Instead, only the logical information and the concrete record values are of interest.

Recovery itself is performed as described in the previous sections. There exists a dedicated master node that provides all joining nodes with the current version of the database. Upon joining of a new node the master executes first all transactions delivered before the view change, then it executes the database extraction function. This function acquires read locks on all relations. Thus, local transactions in their read phase can continue. Incoming write sets are delayed. Once the file is created the read locks are released and write operations can again execute. The file is transferred to the joining node. There, the database is installed and the queued write sets are applied before the joined node accepts requests from users. Postgres-R also handles different failure cases, reacting accordingly if the joining or the master node fail during recovery.

So far, the implementation is rather inefficient since the master is blocked until all data is extracted. Data extraction already takes a couple of seconds for small databases (around 10 tables). This shows the need for a more flexible lock management where locks are released once the corresponding relations are extracted. It also shows that log support will be necessary for bigger databases.

9 Partial Replication

So far, we have assumed full replication. In practice, databases need partial replication: not all data items are replicated and those which are replicated may not have copies at all sites. This also holds for many cluster applications. If the database is very big and/or a transaction typically only accesses data in one partition, then it would be appropriate to store a partition on either one site, some of the sites or all sites in the system. In developing partial replication, several issues have to be considered:

- **Non-replicated Data** If data is only stored locally at one site, access to it should not lead to any message overhead or overhead on remote sites.
- **Replica Control** The replication protocol has to guarantee that all sites which have a copy of a data item receive all updates on this data item. Furthermore, global serializability must be guaranteed despite arbitrary and overlapping data distributions, and the existence of non-replicated data.
- **Subscription** Appropriate mechanisms must be provided in order to subscribe to data (receive a copy of a data item) in a fast and simple manner, and to unsubscribe if copies are no longer needed.
- **Distributed Transactions** If data is not locally available there must be mechanisms to access data on remote sites.

Partial replication is a complex subject, and this thesis can only discuss some of the issues and propose general solutions to the problems without going into much detail. Hence, this chapter is less formal and will only sketch protocol changes and correctness issues. Fault-tolerance is not discussed but is very similar to a fully replicated configuration.

9.1 Non-Replicated Data

Non-replicated data is stored on only one site and is not supposed to be replicated at any other server. Accessing it should not result in any processing costs at remote sites or in any message overhead. This means that only local transactions should have access to this data. Non-replicated data can be easily included in our replica control protocols of Chapter 4. The approach is the same as in traditional solutions: non-replicated data does not need any replica control and is maintained and accessed as in a single-node system.

In the following, we assume that a transaction T_i submitted at node N can access any non-replicated data local at N and any replicated data. All access to non-replicated data is performed during the read phase. Furthermore, we call a transaction *non-replicated* if it only accesses non-replicated data. Otherwise we call it *replicated*. The following description holds both for the protocols with deferred updates and shadow copies.

9.1.1 Non-Replicated Data in the SER Protocol

We enhance the standard SER protocols of Figures 4.1 and 4.5 to the SER-NR protocol in the following straightforward way:

I. *Local Read Phase:*
   3. *Non-replicated data operations:* Acquire a read/write lock for each read/write operation $r_i(X)/w_i(X)$ on non-replicated data item X and perform the operation on the current version of the data item.

That is, we apply strict 2-phase locking to non-replicated data. Conflicts lead to the blocking of the requesting transaction. As a side effect, the protocol can deadlock. This is the only case that a transaction can abort due to non-replicated data. However, as with the RIW locks used for shadow copies, such deadlocks only occur in the read phase (since non-replicated data is only accessed during the read phase) and can be resolved locally. The arguments are the same as in Lemma 4.2.2.

The standard SER protocols considering only replicated data guarantee 1-copy-serializability because conflicting transactions are always serialized according to the total order of write set delivery. In SER-NR, transactions can now also be serialized due to conflicts on non-replicated data that are not seen globally. We will show that despite these local conflicts replicated transactions will still be serialized according to the total order.[1]

Lemma 9.1.1 *The SER-NR protocol produces locally serializable histories at each node.*

Proof Since the original SER protocols produce serializable histories at each node and the enhancement to SER-NR represents a strict 2-phase locking protocol on non-replicated data, the extended version is also serializable. □

Theorem 9.1.1 *The SER-NR protocol is 1-copy-serializable.*

Proof We must show that the global history, i.e., the union of the local histories is 1-copy-serializable. Assume this is not the case and there is a serialization graph of a global history that contains a cycle. In this cycle, there must be at least two replicated transactions, otherwise one of the local graphs would have a cycle which is not possible according to the previous Lemma. Let T_1 and T_2 be any pair of replicated transactions in the cycle, so that there is no other replicated transaction on the path $T_1 \rightarrow \ldots \rightarrow T_2$. We can distinguish several cases:

1. T_1 and T_2 have a direct conflict, i.e., there are no transactions in between T_1 and T_2.
   a.) T_1 and T_2 conflict on replicated data. Then WS_1 is delivered before T_2 according to the original SER protocols.
   b.) T_1 and T_2 conflict on non-replicated data. Since T_2 requests the lock in its read phase and will only receive it once T_1 has committed, WS_1 is delivered before WS_2.
2. T_1 and T_2 conflict indirectly, i.e., there is a path $T_1 \rightarrow T_{i1} \ldots T_{ij} \rightarrow T_2$. By the choice of T_1 and T_2 all transactions between T_1 and T_2 are non-replicated. Since any of these transactions only conflicts with transactions local at the same site, all transactions including T_1 and T_2 must be local at the same node N. Since the conflict between T_{ij} and T_2 must be on non-replicated data, T_2 is still in its read phase when the conflict occurs and hence, can only finish its read phase once all preceding transactions including T_1 have committed (due to strict 2-phase locking of all data involved). Hence, WS_1 is delivered before WS_2.

[1]The proof is similar, but not identical to the one in [Alo97]. The difference is that we do not rely on order preserving serializability [BBG89] or the requirement that replicated transactions are executed serially.

This means that for any pair T_1 and T_2 of replicated transactions, if T_1 is serialized before T_2 then WS_1 is delivered before WS_2. Hence, there cannot be a cycle in the graph. □

9.1.2 Non-Replicated Data in the CS and SI Protocols

The extensions for the CS and SI protocols are very similar and do not require significant changes to the original protocols. For the CS protocol of Figure 4.3 we have:

I. *Local Read Phase:*

3. *Non-replicated data operations:* Acquire a read/write lock for each read/write operation $r_i(X)/w_i(X)$ on non-replicated data item X and perform the operation on the current version of the data item. Release the read lock after the operation if X will not be written later on.

The SI protocols of Figures 4.4 and 4.7 have the following extension:

I. *Local Read Phase:*

3. *Non-replicated data operations:* For each non-replicated read operation $r_i(X)$, reconstruct the version of X labeled with T_j where T_j is the transaction with the highest $TS_j(EOT)$ so that $TS_j(EOT) \leq TS_i(BOT)$. For each non-replicated write operation $w_i(X)$, acquire first a write lock. Then perform a conflict test. If the current version of X is labeled with T_j and $TS_j(EOT) > TS_i(BOT)$ then abort T_i, else perform the operation.

In both cases, serializability is not provided, but the same phenomena are avoided as in the case with only replicated data. The argumentation is exactly the same as for the theorems in Section 4.2.3 and 4.2.4.

9.2 Replica Control

The main reason for replicating data to only some and not all of the sites is the overhead of keeping all copies up-to-date. If local transactions of a node N do not access a certain data item, a copy would only induce unnecessary overhead.

In general, the replication overhead for a data item X consists of two parts. First, there is a CPU/disk overhead at each site with a copy of X for applying all updates. Second, there is a communication overhead for propagating the updates. This communication overhead can be divided into the overhead at the sender, the overhead in the network and the overhead at the receivers. Replica control for partial replication must aim in reducing these types of overhead. That is, the less copies a data item has the smaller the overhead should be. We distinguish two possible replica control mechanisms differing in their update propagation: in a receiver driven approach, the updates are sent to all nodes in the system. In a sender driven approach they are only sent to the sites with copies.

We will use the following notation: A transaction T_i local at N has a complete write set WS_i containing all write operations of T_i (respectively all data items T_i has modified). $\mathcal{N}_i$ is the set of nodes that have a copy of at least one data item X contained in WS_i. For the beginning, we assume that the local node N must have subscribed to all data items accessed by a local transaction T_i. We will relax this requirement in the next section.

9.2.1 Receiver Driven Propagation

A simple solution to partial replica control is to still send the entire write set to all sites. On each site, the message is parsed and the relevant data is extracted. That is, each site determines to which data it has subscribed and only applies updates on these copies. The basic characteristics of such an approach are:

- It provides correctness in a straightforward way. Since all sites still receive all messages and all messages are globally ordered, correctness is provided for the same reasons as for full replication.
- The sender does not need any "meta information" about the data distribution, i.e., about what site has copies of which data.
- The performance gain is that a node will only apply those updates in the write sets that refer to data it has subscribed to. If the node has not subscribed to any of these data items it will discard the write set immediately after reception and parsing, without even starting a transaction.
- It is rather simple to implement.

Integration into the SER/CS Protocols The SER/CS protocols can be directly enhanced to support receiver driven partial replica control. The only component to be added is that remote sites must be able to extract the relevant information from the write set. Since the owner of a transaction has copies of all data items accessed by T_i it can detect all conflicts locally and take a correct decision whether to commit or abort the transaction.

Integration into the SI Protocols More care has to be taken in the SI protocol. In SI, the decision on commit or abort is done by each node independently. This is possible with full replication because each node has the entire information about all data items and their state. With partial replication this is no more the case. If a node does not have the copy of one of the data items in WS_i it cannot independently decide whether T_i can commit because the missing writes might cause an abort. The local node, however, is always able to make a correct decision because we require it to have copies of all updated data items. Hence, we enhance the SI protocol in that the owner of T_i multicasts the outcome of its version check to the other sites similar as it is done in the SER/CS protocols. As a results, the owner of a transaction T_i and all nodes that have subscribed to all data in WS_i can check and terminate T_i independently upon its reception, all other nodes must wait for the decision message of the owner.

Replication Overhead The receiver driven approach does not optimally eliminate all overhead. If the underlying network does not provide physical broadcast, a multicast is transformed into a series of point-to-point messages. In this case, the sender of a message regarding T_i has a considerable higher overhead sending the message to all sites than only to those in $\mathcal{N}_i$. This, however, does not play a role in cluster configurations since they usually dispose of a broadcast connection and the sender only has to send one physical message. The network has similar constraints. Here, even if there is a broadcast facility, some network topologies (like a star network with a local switch) have to copy the message on each receiver link. If there are only few nodes in $\mathcal{N}_i$, a lot of network bandwidth is wasted. On the receiver sites, the message must be received and parsed, even if a node is not in $\mathcal{N}_i$. This overhead also appears in cluster configurations and can only be kept small by parsing the write sets on the smallest possible level in the software hierarchy.

9.2.2 Sender Driven Propagation

Sending the updates of a transaction T_i only to nodes in $\mathcal{N}_i$ seems to be the logical and optimal solution. Then, nodes do not receive any message regarding data items they do not own. Furthermore, in the case there is no broadcast medium, the number of physical messages is significantly reduced. However, the implementation of such an approach is quite complex in several regards.

Write Set Composition An important aspect of our protocols is that they bundle all write operations of a transaction into a single write set in order to keep the message exchange per transaction constant. With partial replication, it is now possible that a site has copies of some but not all of the data items updated by a transaction T_i. If a node should not receive any updates of data items it does not have a copy of, the sender has to pack an individual write set for each node $N \in \mathcal{N}_i$. If we compare this to a receiver driven approach, it is not clear which one produces more overhead at the sending site. If broadcast is available, the receiver driven approach is definitely simpler and faster. If point-to-point messages must be sent, it depends on whether it is faster to send the same message to all sites or individual messages to few sites. Creating individual messages for each $N \in \mathcal{N}_i$ can be avoided, if we can send the same (complete) write set to all sites in $\mathcal{N}_i$. In this case, the sender driven approach has the same requirement as the receiver driven approach in that receivers must be able to filter write set messages in order to extract the relevant updates. Still, a node never has to parse an irrelevant write set.

Knowledge about Data Distribution The second problem is that the sender must have some form of knowledge about to whom exactly to send the write set, i.e., which sites have subscribed to which data. This means, each site must maintain a *replication catalog* that contains information about the data distribution. Whenever a site subscribes to a data item or discards its copy the *replication catalogs* of all sites must be updated. As a consequence, subscription must be tightly synchronized with concurrent transaction processing in order to not miss any updates.

Multiple Groups The third problem is the maintenance of multiple groups and the requirement to provide a global serialization order although each message is only sent to a subgroup of sites. Such a scheme needs special support for multiple overlapping groups. In fact, there are group communication systems that allow sites to be member of several groups. Furthermore, some of them provide total ordering semantics that do not only order messages sent within a group but also across group boundaries, i.e., messages are totally ordered although they are not delivered at all sites [GMS91, Jia95, SG97]. Drawback of the multiple group approach is the overhead of maintaining many groups and determining inter-group orderings. Most existing approaches can only cope with rather small number of groups. Since in our configuration each subset of nodes is a group, we will soon reach the limits of the group communication system.

Here again, it becomes obvious how difficult it is to dynamically change the data distribution in a sender driven approach. Assume the group communication system is able to maintain $2^{|\mathcal{N}|}$ groups ($\mathcal{N}$ being the set of nodes in the system). Now assume, a node $N \in \mathcal{N}$ wants to subscribe to a data item X. Before the subscription, X is replicated at a subset $\mathcal{N}_X \subset \mathcal{N}$ and $N \notin \mathcal{N}_X$. Furthermore assume, transaction T_i concurrently wants to update X (and no other data item) and hence, sends the write set to $\mathcal{N}_i = \mathcal{N}_X$. This write set must be delivered and executed before N can subscribe to X since N does not receive the update. Furthermore, once N has subscribed to X, any transaction must send the write set to $\mathcal{N}_X \cup N$. This would not be a problem if we maintain a group on a data item basis (i.e., N joins the group $\mathcal{N}_{\mathcal{X}}$). However, this would mean that we have a group for each data item in the system and for all combinations of data items. Since there are probably many more data items than nodes in the system many of these groups would have the same set of nodes and group maintenance would be even more complex. Hence, it seems more reasonable to only maintain $2^{|\mathcal{N}|}$ groups. Then, however, a transaction who accesses the data item X has to send the updates to another group once N has subscribed to X. How current group communication systems coordinate subscriptions and concurrent transactions is unclear in most approaches.

Traditional Replica Control Most traditional replica control mechanisms support sender driven partial replication fairly well as long as the data distribution is static. The only requirement is that the sender must have knowledge of who has subscribed to which data. Messages only need to be sent to the sites with copies without any ordering requirement since concurrency control is done independently and does not depend on a pre-order of the transactions. Furthermore, sites only receive relevant information and do not need to filter messages since each operation is sent in an individual message to only those sites with a copy. Note however, that these two characteristics – the serialization order is determined by agreement of the participating databases and messages are sent per operation – are exactly the reasons for the poor performance of the traditional solutions. Furthermore, also traditional protocols need a multicast primitive (although without ordering) because sending individual messages to every site with a copy can be very time consuming at the sending site and even impractical if there exist many copies. Last, some quorum protocols are very inflexible in regard to changes in the data distribution. When quorums are determined by some form of logical configuration (grid, trees etc.) creating and deleting copies require to restructure the configuration delaying concurrent transaction processing.

As a consequence, if we want to avoid these drawbacks of the traditional approaches and still use a sender driven approach we need a better group communication support than currently provided. Thus, the rest of the chapter assumes a receiver driven solution.

9.3 Subscription

Load balancing and changing user behavior requires a flexible partial replication scheme in which sites can subscribe to data very fast and discard copies whenever they are no more needed. Subscription to and unsubscription from data can be initiated externally (e.g. by a database administrator submitting an appropriate command), or internally (e.g. if the system includes an automatic load balancing component). It may not be confound with the creation and deletion of the data item itself. As a result, we can distinguish four maintenance operations on a data item X:

- `Create(X)` creates a new data item X. With full replication, the `create` operation must be executed at all sites in order to produce copies at all sites. Using partial replication, the operation is only executed locally at the node N at which it is submitted. N is then the master of X.
- `Delete(X)` deletes a data item X. Deleting means that all existing copies must be deleted. Hence, the delete operation must be sent to all sites and executed as a normal write operation. In some applications it might be reasonable that only the master of a data item is allowed to initiate its deletion.
- `Subscribe(X)` creates a copy of an existing data item X at the subscribing node N. A peer node N_p that has already a copy of X must transfer the data to N. For instance, the master can be the peer node. Further details are described below.
- `Unsubscribe(X)` is executed locally and discards the local copy of a data item X. Care has to be taken that there is at least one copy of X in the system unless X deleted. To achieve this, the master might be forbidden to issue an `unsubscribe`.

Synchronized Copy Transfer When a node N subscribes to a data item X, it can be provided by any peer node in the system that has already a copy of X. As with recovery (see Chapter 8), transferring the copy of the data item must be coordinated with parallel transaction processing so that all copies have the appropriate values. That is, if there is a concurrent transaction T_i updating data item X, T_i's updates are either reflected in the transferred copy or applied by N itself.

We achieve this with the same mechanism as described in Chapter 8. A subscription is a special form of transaction that is serialized in regard to concurrent transactions according to the total order of the group communication system. The subscribing node N multicasts a `subscribe(X)` request to the group using the total order service. The delivery of this request represents the synchronization point for the data transfer.

Whenever a group member receives the request, it determines deterministically whether it is the peer node N_p. If it is the peer node, it transfers the version of X that includes all updates of transactions that have been delivered before the `subscribe` request but no update of a transaction delivered after the request. How this is exactly done depends on the replica control protocol (SER/SI). A detailed description can be found in Section 8.5. Remind, that as long as N does not have a copy of X it discards all updates on X that are included in remote write sets. N stops discarding these updates once the `subscribe` request is delivered at N itself. From then on it enqueues these updates and applies them in delivery order once the copy of X is installed. After that, N can start its own transactions on X.

Note that using this mechanism the subscribing node does not need to know who already has a copy of the data item since the request is multicast to the entire group.

Traditional Replica Control The traditional ROWAA approach with distributed locking uses a similar subscribe/unsubscribe technique [BHG87]. Since this approach is sender driven, each site maintains a replication catalog describing the data distribution. As in our scheme, a subscription is a special form of transaction. It reads the current value of the data item from one of the other sites (acquiring a read lock) and installs it locally. Furthermore, the subscribing transaction updates all replication catalogs to reflect the new copy. The access to the replication catalog is controlled by locks in order to synchronize with concurrent transaction processing. As a result, since all normal transactions acquire read locks on the local replication catalog and a subscribing transaction acquires write locks on all replication catalogs, transaction processing is delayed during the time of a subscription. In contrast, our approach does not block concurrent transactions.

As mentioned in the previous section, some forms of quorums have further restrictions. If quorums follow a tree structure [AE90] or a grid [Mae85, CAA90], adding new copies or excluding copies require the reconstruction of the configuration. This might be time consuming and further delay transaction processing.

9.4 Remote Access

So far, we have assumed that a transaction only accesses locally available data. If a transactions wants to access a data item that is not available locally, there are three possibilities.

First, the transaction can be rejected as we assumed so far. This is undesirable since it lacks transparency. If we assume that the data distribution can change very fast, we cannot expect a user to know that a data item is not replicated locally. Second, the site can subscribe to the data item on the fly. Since subscription is time consuming (it must run in the context of its own transaction and be synchronized with the rest of the system) the requesting transaction can be blocked for significant time. Furthermore, if this data item is accessed only rarely, the overhead of subscribing, performing the operation itself and unsubscribing (or maintaining an unused copy) might be not acceptable. Therefore, partial replication should offer a third option, namely allowing to access remote data, i.e., executing parts of the transaction at remote sites.

Remote access is needed during the local read phase. If a transaction T_i accesses data for which the local node N has no copy, N contacts a remote site N' in order to perform the operation. If it is a read operation N' returns the read data to N which simply forwards it to the client. If it is a write operation N' returns the updated data version to N where it is stored. Since N has not subscribed to the data item and does not see conflicting access, the locking information is maintained at N'. As a consequence, N cannot decide independently whether T_i will succeed or must be aborted. Instead, all sites providing remote data must agree on the outcome introducing a 2-phase commit.

If N knows exactly which other site in the system has a copy, it can contact the node directly. If this information is not available, N can simply multicast the request to the entire group (without any ordering guarantees). Then, any node that has subscribed to the data item can response. For simplicity, this can always be the master node.

9.4.1 Remote Access in the SER/CS Protocols

Instead of giving a lengthy protocol description for the SER protocols with remote access, we explain the key steps with the example in Figure 9.1. For simplicity of the figure we assume that nodes know where copies reside. The dashed lines represent point-to-point messages, the solid lines are multicast messages. Point-to-point messages do not require any ordering. Transaction T_i starts at site N_2. It wants to read a data item X that is not stored at N_2 but at N_1. It sends a read request to N_1 where the transaction is registered as a subtransaction, the lock is acquired, the read is performed and the response is returned to N_2. The same procedure is repeated with data item Z at N_3. T_i has now finished its read phase, and N_1, N_2 and N_3 build the set of nodes $\mathcal{R}_i$ that have registered T_i. N_1 now multicasts the write set in total order to all sites. The multicast implicitly starts the 2-phase commit among all sites in $\mathcal{R}_i$. When WS_i is delivered at N_2 no decision can yet be made since N_1 must either wait for pre-commit messages from all sites in $\mathcal{R}_i$ or for at least one abort message. In the given example, N_3 votes for commit and sends a pre-commit message to N_2 since there are no conflicts. At site N_1 a write set WS_j is delivered before WS_i conflicting with T_i's read operation. As in the original SER protocol this leads to an abort of T_i and N_1 sends an abort message to N_2. As a result T_i must be aborted everywhere and N_2 multicasts the corresponding abort message. If there had been no abort at N_1, N_1 would also have sent a pre-commit to N_2 which in turn would have multicast a commit. N_4 is the example of a node that receives all messages (and applies them) but is not involved in the voting since it does not have a subtransaction of T_i. Note that if the shadow copy approach is used remote writes during the read phase might lead to distributed deadlocks adding extra complexity.

The correctness of the protocol is intuitively clear. The read phase can now be distributed among several sites which all might abort the transaction upon arriving write sets. All nodes that perform a read phase decide in the same way about the fate of the transaction as the local node did in the original SER protocol. The final voting guarantees that all agree on the same outcome. For all other nodes the correctness of the original SER protocol applies.

Cursor stability works in the same way as SER. Only difference is that most read locks will only be short and released after the operation.

Comparison with Traditional Replica Control Although this mechanism seems to reintroduce all disadvantages of traditional solutions, this is not quite true. First, the 2-phase commit does not take place among all sites having copies but only among those sites who have performed an operation during the read phase. This group will probably be much smaller. Second, the votes are not sent after the complete execution of the transaction but during the lock phase, hence much earlier in the lifetime of the transaction as in traditional protocols.

9.4.2 Remote Access in the SI Protocol

In SI, only conflicts among write operations are resolved. Still, if the owner N of a transaction T_i does not have copies of all data items updated by T_i it can not perform a complete version check on its own. The question is who builds the set of nodes that can decide together about the fate of a transaction. In the

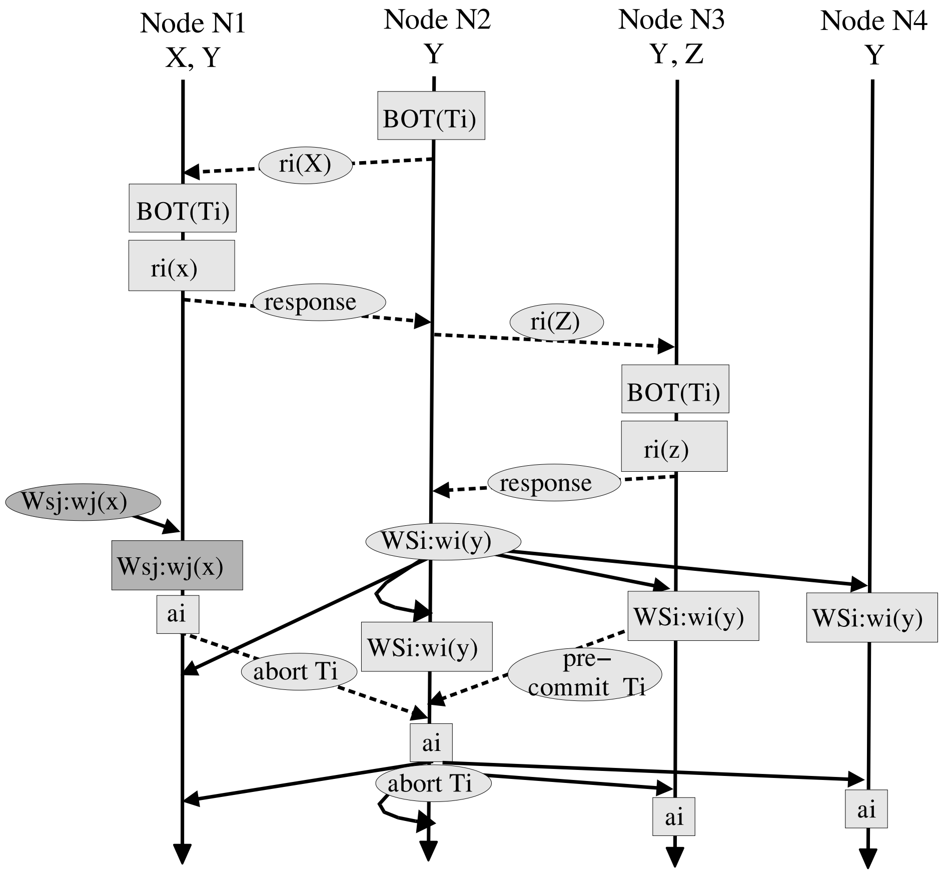

Figure 9.1: Example execution with the SER protocol and remote access

shadow copy approach this is obvious. If the owner N does not have a data item that T_i wants to update it will contact a node N' during the reading phase to perform the update. As a consequence, we can select all the nodes in $\mathcal{R}_i$ as the set of nodes to decide on T_i's outcome. Thus, progression is the same as in the SER protocol with remote access. Remote requests for T_i are sent during the read phase to remote sites and executed there (the owner includes the BOT timestamp in the request). This set of sites builds $\mathcal{R}_i$. Upon delivery of WS_i, all nodes in $\mathcal{R}_i$ perform version checks on all write operations they have copies for. If T_i passes the checks at node N', N' sends a pre-commit message to the owner otherwise it sends an abort message. Nodes without subtransactions do not perform version checks but simply wait (or start executing write operations optimistically as in the SER protocol). If all nodes in $\mathcal{R}_i$ decide on commit the owner multicasts a commit message, otherwise it multicasts an abort.

With deferred writes it is possible that T_i updates a data item for which none of the nodes in $\mathcal{R}_i$ has a copy. Still, it is sufficient that only the nodes in $\mathcal{R}_i$ vote. This can be seen by having a closer look at the "first committer wins" rules of snapshot isolation. Snapshot isolation aborts one of two concurrent writes only because a transaction might have read the data item before writing it and it wants to avoid lost update. In the case of blind writes this abort is, in fact, not necessary. Instead both transactions can be serialized in any order. Since only sites in $\mathcal{R}_i$ have read data, an update of a data item not copied at any site in $\mathcal{R}_i$ can only be a blind write and need not be checked. Hence, SI with deferred writes is identical to SI on shadow copies in regard to the voting phase.

The modified SI protocols still avoid the same phenomena as the original protocols for the same reasons.

9.5 Replication Catalog

Using the receiver driven approach, there is no real need for any “meta information”. Nodes do not need to know the exact data distribution in the system since write sets and remote access requests can be multicast to all sites. Furthermore, nodes do not even need to know exactly which data items exist in the system. When a transaction submits an operation on a data item that is not known locally, the node simply multicasts a remote request to the group.

Not maintaining any meta data, however, has some drawbacks. First, it is not resistant to wrong input. If a user requests access to a data item that does not exist at all, the local node will still submit a remote request but will not receive an answer from any node. Second, remote data access is probably faster and simpler if the sender knows which sites have copies and contacts one of them directly instead of multicasting the request to all sites. Finally, nodes cannot automatically and dynamically subscribe to data upon changes in the workload as it would be needed for an internal load balancing component, since they do not know which data exist. Instead subscription can only be initiated explicitly by an external command.

Therefore, we suggest to maintain a replication catalog at each site that stores two types of global information: the database schema, i.e., the schema information of each data item in the system, and the data distribution, i.e., which node has subscribed to which data. Catalog maintenance is fast and simple. It only requires that also `create` and `unsubscribe` operations are multicast in the entire group. This can happen asynchronously because the information need not be strictly up-to-date (in contrast to the synchronous maintenance requirement in a sender driven approach). We assume this overhead to be not very high, because this information will not change often compared to normal transaction throughput.

The replication catalog is used in the following way:

- Whenever a user references a data item that does not exist locally the transaction checks the replication catalog. If the data item is not registered in the catalog the transaction is aborted because the user has sent an incorrect request. If it is registered one of the sites with a copy is selected for remote access.
- Whenever the system detects unbalanced execution, less loaded nodes look up the replication catalog to subscribe dynamically to new data items.

9.6 Replication Granularity in a Relational Database

So far we have only referred to abstract data items as the granularity of replication. The granularity in a real system, however, depends on the underlying data model and is not automatically defined. Also, different granularity levels result in different complexity for data maintenance.

In a relational system, data is divided into tables, and each table consists of several records. Each record itself consists of several attributes. Usually, tables are seldomly created or deleted. Furthermore, once a table is defined, the attributes of its records are defined. In contrast, records are usually often inserted, updated and deleted.

Table Replication A table represents the natural granularity for replication. That is, a site subscribes to an entire table with all its records. With this, data maintenance is very fast and simple:

- *Remote access:* Since SQL statements explicitly declare all accessed relations in the `from` clause, a node can immediately detect whether it has a local copy of the relations or a remote access is required.
- *Write sets:* Each write set contains a list of all updated relations. Upon delivery of the write set a node checks which tables exist locally and extract only the information related to these relations.

Only problem are SQL statements that modify records of one table R and contain a search on a second table R'. These SQL statements cannot be included in write sets since they cannot be executed at sites that have subscribed to R but not to R'. For these statements, only the physical records can be sent.

Horizontal Replication If tables are very large or contain data with different access frequency, it might be desirable to subscribe to smaller data sets. Using horizontal replication, a node N can subscribe to a subset R_C of the records in a given table R. All records that fulfill the condition C are replicated at N. Th condition C is expressed in form of a `select` statement with a specific `where`-clause. The replication catalog then contains an entry indicating that node N has subscribed to table R with condition C. Although horizontal replication provides sites with the optimal number of records, it increases complexity.

Assume a transaction wants to execute a query Q defined on table R. A node N having subscribed to R_C can only execute Q correctly if all records of R fulfilling Q's search criteria are contained in R_C. If this is not the case the execution of Q at N will return an incomplete result. For instance, assume the subscription criteria C of N and the query Q are as follows:

C: `SELECT * FROM stud-table WHERE dep = ''Computer Science''`
Q: `SELECT * FROM stud-table WHERE age > 25.`

In this case, N will only return computer science students that are older than 25. As a result, upon receiving a query request a node must check the relationship between the query condition and the subscribe condition. In the general case, this check is $\mathcal{NP}$-complete. Furthermore, if none of the sites can return a complete result, results must be joined (eliminating duplicates). Hence, to alleviate the complexity, subscription conditions may only be very simple.

Additionally, with horizontal replication it is not possible to send SQL statements in the write set at all. Assume r is a record not fulfilling C and thus, is not stored at N. Assume further that an `update` statement at node N' changes r in such a way that the new version fulfills C. Hence, r must be installed at N. This can only be done by exclusively sending the new tuple version but not by executing the statement at N.

Vertical Replication Using vertical replication a site does not subscribe to entire records but only to some of the attributes. This can be desirable if the table contains large attributes (e.g. images). Vertical replication is not as complex as horizontal replication, since usually SQL statements explicitly declare the attributes accessed. Hence, sites can easily determine whether a statement can be executed. Still, it is not possible to send SQL statements in the write set, since they might contain a search on missing attributes.

Traditional Replica Control Traditional approaches found in the literature usually do not discuss the granularity of a data item, but similar constraints apply as in our approach. Quorum approaches have even further restrictions. As discussed in Chapter 2, quorum approaches only update a quorum of copies but require read operations to determine which copy in the quorum of read copies has the latest version. In these schemes, maintaining version numbers and determining the latest values can be cumbersome. To illustrate the problem, assume a table R with three tuples X, Y and Z, and three nodes N_1, N_2 and N_3 having subscribed to the table. Furthermore assume, that all tuples are initially written by transaction T_0 and both read and write quorums consist of two sites. Now transaction T_1 updates X on N_1 and N_2, T_2 updates Y on N_2 and N_3, and T_3 updates Z on N_3 and N_1. With this, none of the sites has the latest value of all tuples. If a query wants to read the entire table and it accesses N_1 and N_2 it must read X and Z from N_1 but Y from N_2. This means that even if sites subscribe to entire tables, version numbers must be assigned to tuples and not to tables and queries must compare the results of the different sites on a tuple basis. If sites subscribe to subset of tables, this becomes even more complicated.

In commercial systems, eager update everywhere approaches replicate on a table basis. Lazy approaches use horizontal and vertical subscription. In this case, local queries will simply return incomplete and possibly stale results.

9.7 Partial Replication in Postgres-R

Postgres-R in its current form, provides a simple partial replication component based on the SER protocol [Bau99a]. By integrating partial replication into Postgres-R many of the problems and issues stated above have become obvious. While for many of the solutions proposed in this chapter one seemed much more appropriate than its alternatives from a conceptual point of view, we realized rather fast by looking at a concrete system that their implementation is complex and cumbersome. Examples are the type of update propagation and the replication granularity.

Postgres-R currently uses are receiver driven approach and replicates on a table basis. It provides two new commands at its user interface to subscribe to and unsubscribe from specific tables. Hence, subscription can so far only be done externally. For simplicity, Postgres-R implements a strict master approach. The creator of a table (master) is the only one allowed to delete a table. Furthermore, the master provides subscribing nodes with the table. Postgres-R allows remote read operations. The read operation is multicast to all sites but only the master of a data item will perform the operation and return the result. Read operations on more then one table can only be executed if the local node or one of the table masters has subscribed to all the tables. Distributed joins are not yet possible. A remote read automatically induces a 2-phase commit protocol at the end of the transaction. Remote write operations are not yet possible. An extra replication catalog is maintained to protect against wrong user input. If a user requests access to a table that does not exist locally Postgres-R looks in the replication catalog whether the table exists globally. Only then it submits a remote request, otherwise the transaction is aborted.

Most features needed for partial replication are included in the replication manager for efficiency reasons. The replication manager maintains the replication catalog in form of a system table, and updates it upon all incoming `create table, delete table, subscribe table` and `unsubscribe table` commands. It also extracts all relevant information from the write sets and discards the rest. Only if a write set contains at least one table copied at the local node a remote backend is activated.

With this, Postgres-R provides a nearly complete partial replication component. What is missing is are careful performance study to analyze the performance trade-offs between full and partial replication and the different partial replica control methods.

10 Conclusions

10.1 Summary

This thesis is centered around the topic database replication. Its approach has been motivated and driven by the advances in the development of cluster databases and their specific needs in terms of high throughput, low response times, flexible load balancing, data consistency and fault-tolerance. Eager update everywhere replication appears as an elegant mechanism to achieve these goals. Unfortunately, current solutions seem not to be able to provide sufficient support. Most important, replication must provide performance and scalability – characteristics which traditional approaches often lack. Furthermore, the mechanisms must be simple to implement and to use in order to be an attractive alternative to existing lazy replication implementations. The aim of this thesis has been to develop and implement such an eager update everywhere replication tool. This goal has been accomplished in three steps. First, we have built a theoretical framework, including a suite of different replica control protocols and their correctness proofs. In a second step, the feasibility of the approach has been validated both by means of a simulation study and the integration of the approach into an existing database system. A third step has evaluated further important issues like recovery and partial replication.

A Suite of Replication Protocols By analyzing the advantages and limitations of current solutions it was possible to extract a couple of key concepts and techniques that we think are essential for providing an efficient cluster solution. The basic mechanisms behind our protocols are to first perform a transaction locally, deferring writes or performing them on shadow copies. At commit time all updates (the write set) are sent to all copies in a single message. Delaying the propagation of updates makes it possible to keep the number of messages per transaction small. The write set is sent using a total order multicast provided by group communication systems. By obeying this total order whenever transactions conflict, the global serialization order can be determined individually at each site. Upon reception of the write set each site (including the local site) orders and executes conflicting write operations in the order of write set delivery. In regard to read/write conflicts our protocols provide various mechanisms implementing different levels of isolation. As a result, the protocols do not require a 2-phase-commit protocol and avoid global deadlocks. Furthermore, two levels of fault-tolerance are offered by choosing between two reliability degrees for message delivery. Finally, in order to fasten execution at the remote sites, the write set can contain the physical values of the changed data items which can be simply applied without reexecuting the operation.

Simulation Study Using a detailed simulation system we have evaluated the entire range of proposed algorithms and studied the implications of different hardware configurations and workloads. It has been shown that our approach has better performance than traditional approaches for most workloads and configurations. Furthermore, by providing a whole family of protocols, the system can adapt to various extreme situations without degradation. For instance, a completely fault-tolerant, fully serializable protocol can be used if communication is fast, and if the system load and conflict rates are low. If conflict rates are high, lower levels of isolation alleviate the problem. If the basic communication is costly or if many nodes are in the system, lower levels of fault-tolerance will still provide good response times.

Postgres-R and its Performance We integrated one of the proposed algorithms into the database system PostgreSQL and built the replicated database architecture Postgres-R. With this implementation we have been able to evaluate the complexity issues involved in integrating such a tool into a system with specific constraints. It proved that the framework is flexible enough to be adapted rather well to these constraints and we believe that it can be integrated in a similar way into a wide range of database systems without too much overhead and changes to the original system.

The evaluation of the system in a cluster configuration verified the results of the simulation study and showed, that Postgres-R is stable even at high throughputs and high update rates. Furthermore, it can provide good performance and scalability for a wide range of workloads. It also allows for flexible load balancing since differently loaded nodes do not have a strong impact on each other.

Recovery and Partial Replication Our solution to recovery fits smoothly into the proposed framework and can easily be adjusted to the specifics of the underlying database system. It provides a flexible node management since nodes can leave and join the system online without big impact on the concurrent transaction processing in the rest of the system. Care must be taken that the joining node receives an accurate state of the database without missing any updates. A prototype recovery component exists in Postgres-R.

Partial replication and its efficient implementation is especially crucial in large databases or high throughput systems. We propose a receiver driven scheme that involves little management overhead. All updates are still sent to all sites which are responsible to extract the relevant information. This keeps the overhead at the sender small and allows to still use the total order as a global serialization scheme. The remote sites only apply updates to data items that have copies for. Using a flexible subscribe/unsubscribe system ,the data distribution can be changed online without interrupting transaction processing. This is essential for dynamic load balancing. We have also analyzed access to non-replicated data (for which communication is not necessary) and introduced distributed transactions in case data is not copied locally. Only in the case of distributed transactions an optimized 2-phase commit is necessary.

10.2 Ongoing Work

We have made an effort to present the thesis as a logical unit of work – from the problem statement over the development of a theoretical framework to its implementation and evaluation. Besides the issues discussed here, we have been working on related problems. Most of this work has been embedded in the DRAGON project.

Group Communication and Application Semantics The work presented in this thesis uses multicast primitives as they are currently provided by group communication systems. In general, we believe that group communication systems are a very helpful tool to support database replication. Only if powerful communication primitives are used, eager replication can be implemented efficiently. However, the primitives provided must be applied in a careful manner to avoid unnecessary overhead, and existing group communication primitives do not exactly offer the semantics that are needed for database replication. FIFO and causal order are mostly unrelated to the needs of database systems. The total order, although sufficient to globally order transactions, is in fact too strong since non-conflicting write sets could be delivered in any order.

Instead, what is needed is a way to combine database semantics and group communication primitives in such a way that messages are only ordered when the database system demands it. As a first step in this direction we have developed an optimistic protocol that *overlaps transaction processing with the communication overhead* in order to hide message delays [KPAS99a, KPAS99b, PJKA00]. The motivation has been a new approach to implement atomic broadcast which reduces the average delay for message delivery to the application [PS98]. The protocol takes advantage of the fact that, in clusters, messages normally arrive at the different sites exactly in the same order. Roughly speaking, this protocol considers the order messages arrive at each site as a first optimistic guess, and only if a mismatch of messages is detected, further coordination rounds between the sites are executed to agree on a total order. We have developed this idea further, and show how applications can take full advantage of the optimistic assumption by overlapping the delay incurred to determine the total order with the processing of delivered messages. The basic idea is that the communication system delivers messages twice. First, a message is preliminary delivered to the database system as soon as the message is received from the network. The transaction manager uses this tentative total order to determine a scheduling order for the transaction and starts executing the transaction. While execution takes place without waiting to see if the tentative order was correct, the commitment of a transaction is postponed until the order is confirmed. When the communication system has determined the definitive total order, it delivers a confirmation for the message. If tentative and definitive orders are the same or there are no conflicts, the transactions can commit. Otherwise the wrongly ordered transactions have to be undone and reexecuted in the correct definitive order.

With this approach we can overlap communication and transaction processing, providing short response times without relaxing transaction correctness. Even if messages are not spontaneously ordered by the network, the approach is applicable if conflict rates are small (since disordered transactions are only aborted if they conflict). In this regard, the optimistic approach also seems interesting in environments in which determining the total order is generally time consuming. The same holds for uniform reliable delivery whose performance decreases with increasing number of nodes. Using such an approach it might be possible to extend our approach to systems outside the range of cluster computing.

Classification of Replication Protocols Replication is not only important in databases but also in distributed systems in general. As a consequence, there exist an enormous amount of different approaches, many of them having similar but not identical assumptions and underlying models. Chapter 2 provides a short overview of only some of the approaches and is far of being complete. In the context of the DRAGON project, we have started to provide a framework to compare replication solutions. In [WPS$^+$00b], we classify replication protocols by distinguishing different phases in the replica control mechanisms. In [WPS$^+$00a], we provide a parameter based classification.

10.3 Outlook

While this thesis has been an attempt to provide an eager update everywhere solution that provides efficiency, fault-tolerance and correctness for cluster configurations, there are many more problems to consider and to investigate.

- While first solutions for recovery and partial replication have been presented in this thesis, there are still plenty of open questions and need for further development. Clearly, the performance of the suggested approaches and possible alternatives has to be evaluated. So far, only rather simple forms of recovery and partial replication are implemented in Postgres-R. In regard to recovery, further investigation is required to handle complex failure scenarios. These include topics like the efficient joining of multiple sites (as they occur after network partitions), transparent handling of failures during recovery, or recovery from a complete system crash. There might also be more efficient solutions for the data transfer, especially solutions that further decrease the impact on concurrent transaction processing.
- Concerning partial replication there is the need for group communication primitives that efficiently and adequately support multiple groups. Another question is authorization. Data might only be partially replicated because some sites are not authorized to see specific data items. If the sender does not pack individual packages for all receivers but sends the same entire write set to all relevant sites, it must be either possible to delete irrelevant data at the receiving site before any unauthorized person can access it, or some crypting technique has to prevent access to unauthorized data. A further important topic, that has only been shortly discussed in this thesis, is the granularity of data items that can be replicated. Looking at relational databases, it would, in theory, be desirable to allow sites to subscribe to arbitrary subsets of tables. Such an approach, however, requires sophisticated mechanisms to determine which sites are able to perform what kind of operations.
- Clearly, there are architectures and configurations, in which lazy replication is the better choice: if communication is extremely costly and cannot be hidden behind optimistic transaction processing, if conflict rates are extremely low, if inconsistencies can be resolved easily, if fault-tolerance is not an issue, or if primary copy is not a too big limitation, the disadvantages of lazy replication do not apply and its performance in terms of response times will win the race. It would be interesting to analyze the exact trade-offs between lazy and eager replication and to be able to quantify the decisive factors: when exactly is communication too costly for eager replication, when are conflict rates small enough for lazy replication, etc.
- This thesis has focused on cluster configurations with a fast local area network. Current networks, however, cannot simply be divided into LANs and WANs anymore. More and more new network types are evolving. Mobile and ubiquitous computing dissolve the static configuration of current networks and introduce new moving devices storing and manipulating data in a different way. These new networks present new challenges for data management in general and replication in particular.
- In regard to integrating application semantics and multicast primitives, overlapping transaction execution with communication is only one solution and many more possibilities might exist. An interesting first step into a much tighter integration is presented in [PS99]. Here, messages are only ordered if the application requires it. In order to achieve this, the group communication system asks the application whether a set of messages must be ordered or not. A further step would be to directly include the specific application semantics – e.g., the serialization order – into the multicast primitive. Especially, if determining the total order is time consuming and optimistic assumptions do not hold, this could be an attractive alternative.

Bibliography

[AAE+96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in the workflow contexts. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 574–581, New Orleans, Louisiana, February 1996.

[AAES97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proc. of Europ. Conf. on Parallel Processing (Euro-Par)*, pages 496–503, Passau, Germany, August 1997.

[ABKW98] T. A. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 484–495, Seattle, Washington, June 1998.

[ACL87] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proc. of the Int. Conf. on Software Engineering*, pages 562–570, San Francisco, California, October 1976.

[Ady99] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations of Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[AE90] D. Agrawal and A. El Abbadi. The tree quorum protocol: an efficient approach for managing replicated data. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 243–254, Brisbane, Australia, August 1990.

[AES97] D. Agrawal, A. El Abbadi, and R. C. Steinke. Epidemic algorithms in replicated databases. In *Proc. of the ACM Int. Symp. on Principles of Database Systems (PODS)*, pages 161–172, Tucson, Arizona, May 1997.

[Alo97] G. Alonso. Partial database replication and group communication primitives. In *Proc. of European Research Seminar on Advances in Distributed Systems (ERSADS)*, Zinal, Switzerland, March 1997.

[ALO00] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 67–78, San Diego, California, March 2000.

[ANS92] ANSI X3.135-1992. *American National Standard for Information Systems – Database Languages – SQL*. November 1992.

[ARM97] G. Alonso, B. Reinwald, and C. Mohan. Distributed data management in workflow environments. In *Proc. of the Int. Workshop on Research Issues in Data Engineering (RIDE)*, Birmingham, United Kingdom, April 1997.

[Bac99] I. Bachmann. Konfigurationsmanagement in Postgres-R: Recovery und Datenbankkonfiguration. Master's thesis, Department of Computer Science, ETH Zürich, Switzerland, 1999.

[Bau99a] M. Baumer. Integrating synchronous partial replication into the PostgreSQL database engine. Master's thesis, Department of Computer Science, ETH Zürich, Switzerland, 1999.

[Bau99b] W. Bausch. Integrating synchronous update-everywhere replication into the PostgreSQL database engine. Master's thesis, Department of Computer Science, ETH Zürich, Switzerland, 1999.

[BBG89] C. Beeri, P.A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

[BBG+95] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–10, San Jose, California, June 1995.

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

[BG84] P.A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, December 1984.

[BGRS00] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the coordination overhead of synchronous replica maintenance in a cluster of databases. In *Proc. of Europ. Conf. on Parallel Processing (Euro-Par)*, pages 435–444, Munich, Germany, August 2000.

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

[BJ87a] K. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the ACM Symp. on Operating Systems Principles (SOPS)*, pages 123–138, Austin, Texas, November 1987.

[BJ87b] K. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[BJK+97] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton. The Oracle universal server buffer. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 590–594, Athens, Greece, August 1997.

[BK97] Y. Breitbart and H. F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, pages 173–184, Tucson, Arizona, May 1997.

[BKR+99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 97–108, Philadephia, Pennsylvania, June 1999.

[BN97] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufman Series in Data Management Systems, 1997.

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

[Buy99] R. Buyya. *High Performance Cluster Computing*. Prentice Hall PTR, 1999.

[CAA90] S. Y. Cheung, M. Ahamad, and M. H. Ammar. The grid protocol: A high performance scheme for maintaining replicated data. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 438–445, Los Angeles, California, February 1990.

[CHKS94] S. Ceri, M. A. W. Houtsma, A. Keller, and P. Samarati. A classification of update methods for replicated databases. Technical report, Computer Science Department, Stanford University, CS-TR-91-1392, 1994.

[CL89] M. J. Carey and M. Livny. Parallelism and concurrency control performance in distributed database machines. In *Proc. of the ACM SIGMOD Management on Data*, pages 122–133, Portland, Oregon, June 1989.

[CL91] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Transactions on Database Systems*, 16(4):703–746, 1991.

[CP92] S.-W. Chen and C. Pu. A structural classification of integrated replica control mechanisms. Technical report, Department of Computer Science, Columbia University, New York, CUCS-006-92, 1992.

[CRR96] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 469–476, New Orleans, Louisiana, February 1996.

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

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[DGMS85] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[DM96] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[ES83] D. L. Eager and K. C. Sevcik. Achieving robustness in distributed database systems. *ACM Transactions on Database Systems*, 8(3):354–381, September 1983.

[ET89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, 1989.

[FGS98] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA group communication service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[FvR95a] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. Technical report, Department of Computer Science, Cornell University, TR-95-1527, 1995.

[FvR95b] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. Technical report, Department of Computer Science, Cornell University, TR-95-1537, 1995.

[GHOS96] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[GHR97] R. Gupta, J. R. Haritsa, and K. Ramamritham. Revisiting commit processing in distributed database systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 486–497, Tucson, Arizona, June 1997.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the ACM SIGOPS Symp. on Operating Systems Principles*, pages 150–162, Pacific Grove, California, December 1979.

[GLPT76] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared database. In *Proc. of the IFIP Working Conf. on Modelling in Data Base Management Systems*, pages 365–394, Freudenstadt, Germany, January 1976.

[GMS91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

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

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

[GS96] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols in distributed systems. In *Proc. of the Int. Symp. on Fault-Tolerant Computing (FTCS)*, pages 168–177, Sendai, Japan, 1996.

[GSC+83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. R. Ries. A recovery algorithm for a distributed database system. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, pages 8–15, Atlanta, Georgia, March 1983.

[HAA99] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group multicast. In *Proc. of the Int. Symp. on Fault-Tolerant Computing (FTCS)*, pages 158–165, Madison, Wisconsin, June 1999.

[Hay98] M. Hayden. *The Ensemble System*. PhD thesis, Dept. of Computer Science, Cornell University, 1998.

[HSAA00] J. Holliday, R. Steinke, D. Agrawal, and A. El Abbadi. Epidemic quorums for managing replicated data. In *Proc. of the IEEE Int. Performance, Computing and Communications Conf. (IPCCC)*, pages 93–100, Phoenix, Arizona, February 2000.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison-Wesley, 1993.

[Jia95] Xiaohua Jia. A total ordering multicast protocol using propagation trees. *ACM Transactions on Parallel and Distributed Systems*, 6(6):617–627, June 1995.

[JM87] S. Jajodia and D. Mutchler. Dynamic voting. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 227–238, San Francisco, California, May 1987.

[KA98a] B. Kemme and G. Alonso. Database replication based on group communication. Technical Report 289, Department of Computer Science, ETH Zürich, Switzerland, February 1998.

[KA98b] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 156–163, Amsterdam, The Netherlands, May 1998.

[KA99] B. Kemme and G. Alonso. Transactions, messages and events: Merging group communication and database systems. In *Proc. of European Research Seminar on Advances in Distributed Systems (ERSADS)*, Madeira (Portugal), April 1999.

[KA00a] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, September 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, September 2000.

[KB91] N. Krishnakumar and A. J. Bernstein. Bounded ignorance in replicated systems. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, pages 63–74, Denver, Colorado, May 1991.

[Kem97] B. Kemme. Datenbankreplikation unter Verwendung von Gruppenkommunikation. In *9. Workshop "Gundlagen von Datenbanken"*, pages 46–50, Friedrichsbrunn, Germany, May 1997.

[KPAS99a] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431, Austin, Texas, June 1999.

[KPAS99b] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Using optimistic atomic broadcast in transaction processing systems. Technical Report 325, Department of Computer Science, ETH Zürich, Switzerland, March 1999.

[KS93] A. Kumar and A. Segev. Cost and availability tradeoffs in replicated concurrency control. *ACM Transactions on Database Systems*, 18(1):102–131, March 1993.

[KT96] M. F. Kaashoek and A. S. Tanenbaum. An evaluation of the Amoeba group communication system. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 436–448, Hong Kong, May 1996.

[Mae85] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985.

[MAMSA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 56–65, Poznan, Poland, June 1994.

[Mes99] Mesquite Software, Inc. *CSIM18 – The Simulation Engine*, March 1999. http://www.mesquite.com.

[MHL+92] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[MMSA+96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[Moh90a] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions on B-tree indexes. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 392–405, Brisbane, Queensland, Australia, 1990.

[Moh90b] C. Mohan. Commit-LSN: A novel and simple method for reducing locking and latching in transaction processing systems. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 406–418, Brisbane, Queensland, Australia, 1990.

[NT88] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed systems. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 248–262, Toronto, Canada, August 1988.

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

[Ora97] Oracle. *Oracle8(TM) Server Replication, Concepts Manual*, 1997.

[Ped99] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, December 1999.

[PG97] F. Pedone and R. Guerraoui. On transaction liveness in replicated databases. In *Proc. of IEEE Pacific Rime Int. Symp. on Fault-Tolerant Systems (PRFTS)*, Taipei, Taiwan, December 1997.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proc. of the Symp. on Reliable Distributed Systems (SRDS)*, pages 175–182, Durham, North Carolina, October 1997.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of Europ. Conf. on Parallel Processing (Euro-Par)*, pages 513–520, Southampton, England, September 1998.

[PGS99] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. Technical Report SSC/1999/008, École Polytechnique Fédérale de Lausanne, Switzerland, March 1999.

[PJKA00] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the Int. Symp. on Distributed Computing (DISC)*, Toledo, Spain, October 2000.

[PL88] J. F. Pâris and D. E. Long. Efficient dynamic voting algorithms. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 268–275, Los Angeles, California, February 1988.

[PL91] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 377–386, Denver, Colorado, May 1991.

[Ple97] S. Pleisch. Database replication on top of group communication: A simulation tool. Master's thesis, Department of Computer Science, ETH Zürich / EPF Lausanne, Switzerland, 1997.

[PMS99] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 126–137, Edinburgh, Scotland, September 1999.

[Pos98] PostgreSQL. *v6.4.2 Released*, January 1998. http://www.postgresql.org.

[PS98] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proc. of the Int. Symp. on Distributed Computing (DISC)*, pages 318–332, Andros, Greece, September 1998.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *Proc. of the Int. Symp. on Distributed Computing (DISC)*, pages 94–108, Bratislava, Slavak Republic, September 1999.

[PSM98] E. Pacitti, E. Simon, and R. N. Melo. Improving data freshness in lazy master schemes. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 164–171, Amsterdam, The Netherlands, May 1998.

[RGK96] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 207–222, Avignon, France, March 1996.

[Rie99] G. Riedweg. Entwicklung eines Replikations-Managers. Department of Computer Science, ETH Zürich, Switzerland, 1999. Semesterarbeit.

[RNS96] M. Rys, M. C. Norrie, and H.-J. Schek. Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 460–471, Mumbai (Bombay), India, September 1996.

[RST95] S. Rangarajan, S. Setia, and S. K. Tripathi. A fault-tolerant algorithm for replicated data management. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1271–1282, December 1995.

[SA93] O. T. Satyanarayanan and D. Agrawal. Efficient execution of read-only transactions in replicated multiversion databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, 1993.

[SAE98] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 148–155, Amsterdam, The Netherlands, May 1998.

[SAS+96] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 485–494, New Orleans, Louisiana, February 1996.

[Sch86] H. D. Schwetman. CSIM: A C-based process-oriented simulation language. In *Winter Simulation Conference*, pages 387–396, Washington DC, December 1986.

[Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[SES89] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *Proc. of the Int. Workshop on Distributed Algorithms (WDAG)*, pages 219–232, Nice, France, September 1989.

[SG97] A. Schiper and R. Guerraoui. Total order multicast to multiple groups. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, Baltimore, Maryland, May 1997.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[SRH90] M. Stonebraker, L. A. Rowe, and M. Hiroham. The implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[SS93] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 561–568, Pittsburgh, Pennsylvania, 1993.

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

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.

[SW99] R. Schenkel and G. Weikum. Federated transaction management with snapshot isolation. In *Proc. of the Int. Workshop on Foundations of Models and Languages for Data and Objects - Transactions and Database Dynamics (FMLDO)*, pages 139–158, Dagstuhl, Germany, September 1999.

[SW00] R. Schenkel and G. Weikum. Integrating snapshot isolation into transactional federations. In *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, Eilat, Israel, September 2000.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[TP98] O. Theel and H. Pagnia. Optimal replica control protocols exhibit symmetric operation availabilities. In *Proc. of the Int. Symp. on Fault-Tolerant Computing (FTCS)*, pages 252–261, Munich, Germany, 1998.

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

[WPS+99] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. Technical Report SSC/1999/035, École Polytechnique Fédérale de Lausanne, Switzerland, September 1999.

[WPS+00a] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proc. of the Symp. on Reliable Distributed Systems (SRDS)*, Nürnberg, Germany, October 2000.

[WPS+00b] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 264–274, Taipei, Taiwan, April 2000.