PRACTICAL AND CONSISTENT DATABASE REPLICATION

Yi Lin

DOCTOR OF PHILOSOPHY

the School of Computer Science

MCGILL UNIVERSITY

MONTREAL, QUEBEC

DECEMBER 2007

A THESIS SUBMITTED TO MCGILL UNIVERSITY IN PARTIAL FULFILMENT OF THE REQUIREMENTS OF THE DEGREE OF DOCTOR OF PHILOSOPHY

Your file Votre référence
ISBN: 978-0-494-50951-7
Our file Notre référence
ISBN: 978-0-494-50951-7

ACKNOWLEDGEMENTS

I would like to express my greatest gratitude to my supervisor, Dr. Bettina Kemme, for her continued guidance, financial supports, encouragement and patience through all the phases of this research.

Extended thanks to Dr. Ricardo Jiménez-Peris and Dr. Marta Patiño-Martínez, for their lively and fruitful discussions regarding this research.

My sincere appreciation to all my friends and colleagues whom I met in Montreal, Beijing, and Hainan, for their joys and laughs.

Finally, but certainly not least, my deepest appreciation to my family, especially to my mother and my brother, who have been so supportive and loving. Without them, this report would not be the same.

ABSTRACT

Replicating data across different databases has the potential to provide low response times since data can be accessed locally, high scalability since load can be distributed, and fault-tolerance since the data can be accessed as long as one replica is available. A big challenge of database replication is to handle updates such that the entire replicated database appears as if there is only a single logical copy of the data.

The standard correctness criterion for database replication is 1-copy-serializability (1-copy-SE) which guarantees that a replicated database behaves as a non-replicated database with serializability (SE), the highest isolation level for transaction execution. In this thesis, we propose a new criterion, 1-copy-snapshot-isolation (1-copy-SI), due to the popularity of snapshot isolation (SI) over serializability in major database systems. SI allows some non-serializable executions, but it provides better concurrency and can be implemented efficiently. However, current definitions of SI allow for the violation of integrity constraints while commercial implementations of snapshot isolation maintain them. Hence, we define a new isolation level SI+IC which represents the isolation level implemented in current systems. From there, we propose a criterion 1-copy-SI+IC that respects both SI and integrity constraints in a replicated database.

As a second step, we develop a replication solution that provides many features. It provides 1-copy-SI+IC. It is implemented in a middleware between clients and original database system, and thus, does not require changes to the source code of the database system. Despite being at the middleware level, it provides concurrency at the record level, and thus, the same concurrency level as the database system itself. Furthermore, it provides a standard database interface, and thus, is transparent to the application. It also offers fault-tolerance. Finally, it includes protocols that are

able to handle a wide-area environment. This is achieved by a careful choice of communication patterns that keep communication across the wide area network at a minimum.

The approach is implemented within a middleware-based framework that allows for an easy plug-in of replication algorithms. Our solution is carefully evaluated, comparing several design alternatives. Additionally, it is compared against a traditional replication protocol, that is widely implemented in current systems. The evaluation shows that our protocols have very good performance and compare favourably with existing solutions.

ABRÉGÉ

La réplication de bases de données réplique les données dans differentes bases de données dans le but d'offrir des temps de réponse rapides, de meilleures possibilités de croissance et d'extension future, ainsi qu'une meilleure tolérance aux fautes. La problématique de la réplication de bases de données est comment répliquer les données correctement de sorte à ce que le système complet de bases de données répliquées se comporte comme s'il y avait une seule copie de la base de données.

Le critére standard d'exactitude est le "one-copy-serializability". Dans cette thèse, nous proposons un nouveau critère, "one-copy-snapshot-isolation", du à la popularité de "snapshot-isolation", comparativement à la "serializability", dans les principaux systèmes d'administration de bases de données. Le "snapshot-isolation" est une notion plus faible que la "serializability" sur plusieurs aspects, dont l'un est de ne pas garantir les contraintes d'intégritées. Nous proposons par la suite un critère qui inclut les notions de "snapshot-isolation" ainsi que les contraintes d'intégritées. La thèse propose un formalisme basé sur des travaux antérieurs.

Dans un deuxième temps, nous étudions comment développer des protocoles de réplication qui peuvent être utilisés en practique. Quelques protocoles existants imposent de sévères limitations à leurs applications, comme l'identification des transactions en lecture seule à leur commencement. D'autres protocoles ne fonctionnent pas avec des bases de données avec contraintes d'inégritées, une caractéristique très importante des bases de données. Nous proposons une solution qui résoud toutes ces limitations à l'aide d'un ordonnanceur centralisé.

Troisièmement, cette thèse étudie les protocoles de réplication qui fonctionnent bien, tant dans des réseaux locaux que globaux. La plupart des protocoles existants requièrent plusieurs étapes d'échange de message entre differentes bases de données à l'intérieur d'un certain temps de réponse,

et/ou utilisent des messages "multicast" traitées par des systèmes de communication de groupe. Ils ne fonctionnent pas bien dans les réseaux globaux en raison des longs temps d'attentes pour l'envoie de messages. Nous étendons notre approche en utilisant une approche décentralisée. La nouvelle solution garde un nombre constant de message à l'intérieur d'une transaction (un allée-retour de messages). Le probléme de la tolérance aux fautes est aussi discuté.

Pour évaluer notre solution, nous avons developé un ensemble de fonctionnalités qui integre l'implémentation des différents protocoles avec un système de bases de données, PostgreSQL. Les expérimentations ont été éxecutées sur différents systèmes, e.g., réseaux locaux et globaux. Les résultats sont satisfaisants.

En somme, cette thèse présente une solution pratique au problème de réplication de bases de données qui fonctionne bien autant dans des réseaux locaux que globaux. Elle a été implementée dans un système temps réel et les résultats confirment qu'elle est plus efficace que les protocoles existants.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Why database replication

As the Internet grows at a rapid pace, computer applications transit from desktops to the network. Network applications target to support users in different geographical locations. The transition introduces many challenges. For example, how can users access data fast even though they go through the Internet? As another example, how can a system scale to accomodate more and more users, and how can the availability of a system be ensured in case of disasters like 911?

Let's have a look at a concrete example. An online flight booking system such as Expedia is used by many clients around the world. Clients browse information such as prices and itineraries, and make reservations. All information about flights is stored in a database. Suppose there is only one database located in Montreal for the whole system. Although clients in Montreal can enjoy a fast service when they browse flight information, clients in other cities such as London will complain about the slow connection, because all information must be transferred from Montreal to London for display.

Another challenge is to scale the system when more and more clients want to access the service. A single database can only support a certain number of clients. If the number of clients exceeds the threshold, the database will be overloaded and the system will respond to clients extremely slowly.

Furthermore, the database might crash due to overload or other reasons. If this occurs the

system can not respond to client requests until the database is recovered. This can very fast result in financial losses for the company. More seriously, if the database is physically destroyed, then all data stored in the server will be completely lost, which is a disaster for the business.

This thesis aims at providing answers to the above challenges by using replicated databases. A replicated database system is composed of many copies of the database distributed across different sites. Each database, being called a replica, can accept client requests. The database replicas work cooperatively as a single global database system to provide database services to clients at all sites. Database replication can improve both fault-tolerance and performance. In regard to fault-tolerance, since there exist several copies of the database, if one database crashes, data is still available since other replicas can be accessed. The client requests submitted to the crashed database can be redirected to replica(s) remaining accessible.

In regard to performance, database replication can increase the throughput of a system and decrease the response time of individual requests. Many replicaton solutions follow a scheme of Read-One-Write-All (ROWA) [14, 78]. According to ROWA, write operations that update data items are performed at all replicas to guarantee data consistency while read operations only need to be performed at one replica. Since read requests are only executed at one replica, by adding replicas to the system, the capacity in terms of throughput (i.e., number of requests per time-unit) can be increased.

In order to keep response time low, data can be replicated to remote sites so that clients at the remote sites can access the data just locally. In the online flight booking system example above, clients in London can then read data from the local database replica in London. Thus, the clients experience fast response since WAN communication does not occur. With ROWA, reads are as fast as having a single, local database. However, writes trigger a considerable overhead since all replicas need to be updated. This is acceptable if the ratio of reads to writes is high, and has shown to outperform basically all other approaches such as quorums [60].

1.2 Challenge of database replication

One main challenge of database replication is to keep the data copies consistent in the presence of updates. If a client updates a data item, the update has to be propagated to all copies. If clients connected to different replicas submit updates on the same data items, such updates have to be coordinated to guarantee that the data remains consistent. This task is called *replica control.* In the example above, there might be two clients competing for a single flight ticket in a last minute deal. If they make their reservation in different cities and at the same time, the system must guarantee that only one of them succeeds in making the reservation. Otherwise, complex reconcilation techniques are needed, possibly requiring some of the clients to receive some compensation.

Besides data consistency, there are some other challenges that a replicated architecture has to face. For instance, in an ideal replicated database system replication is transparent to clients. The system appears as if there is only one single database. This is particular challenging since clients access the database in the context of transactions [101]. Clients submit their requests in form of transactions consisting of a sequence of read and write operations on the data items in the system. The operations in a transaction are considered a logical unit of work. Either all updates of a transaction succeed and the transaction commits, or none of its updates have an effect and the transaction aborts. Furthermore, although the database system might interleave the execution of different transactions, the concurrency control mechanism of the system isolates them to guarantee there is no improper interference.

Thus, achieving transparency in a replicated system requires that the global execution of transactions is equivalent to an execution over a single logical database. This means replica control must always be coupled with the concurrency control mechanisms of the database, in order to provide correct executions.

1.3 Existing work and their shortcomings

Many replication solutions have been proposed so far. Some replication protocols provide strong consistency meaning that data is consistent at any time. This, however, increases response time

for update transactions because replicas must coordinate their updates before transactions commit. Other protocols only provide weak consistency meaning that data may be inconsistent temporarily though it will be eventually consistent. This provides fast response for writes but read operations may access stale data when they read local data which do not yet reflect updates performed on remote replicas. Some protocols with weak consistency even require to rollback transactions that have already been committed. This complicates the system and exposes to applications a behavior non-existent in non-replicated systems. Section 2.2 will discuss in more detail existing replica control strategies. Many protocols with strong consistency have been recently proposed (e.g., [6, 8, 7, 10, 39, 38, 24, 55, 61, 91]). They guarantee data consistency at any time and provide reasonably good performance. However, these protocols only work well in Local Area Networks (LANs) but not in Wide Area Networks (WANs) because they have a fair amount of message overhead which is unacceptable in WANs. This thesis proposes replication protocols that offer *strong data consistency* with little message overhead.

Moreover, many of the existing replica control protocols ignore how current database systems implement concurrency control. As mentioned above, although databases allow transactions to execute concurrently and access data simultaneously, transactions may not arbitrarily interfere with each other. Instead, different transaction isolation levels have been defined. They refer to the extend to which concurrent transactions may access the same data items. The strongest transaction isolation level is serializability (SE) [14, 97, 110]. With SE, although transactions may execute concurrently, the effect is the same as running the transactions serially one after another. Many databases have concurrency control mechanisms that guarantee SE. Therefore, most replicated systems provide *1-copy-SE*, the extension of SE in a replicated environment. It guarantees that the entire system behaves as if there were only one logical database (i.e., one-copy) providing SE.

However, recently, *Snapshot Isolation (SI)* has emerged as a new isolation level [12]. SI is slightly weaker than SE and has become quite popular. It requires that transactions read data from a snapshot committed at the time point when they start. Furthermore, if two transactions want to update the same data item at the same time, one will be aborted. SI has been adopted by many database vendors such as Oracle, PostgreSQL, Interbase 4 and Microsoft SQL Server 2005. Although not

being as strong as SE as defined in the research literature, SI avoids all isolation anomalies as defined by the industrial ANSI standard [11]. Hence, Oracle and PostgreSQL claim that their SI-based concurrency control mechanisms actually provide SE. Although SI has become popular in industry, little has been done on developing replication solution based on SI. Furthermore, many replication protocols do not consider *Integrity Constraints (ICs)* which are an important feature of relational database technology. Ignoring integrity constraints prevents those protocols from working correctly with databases that have integrity constraints defined. This thesis proposes a new isolation level, *1-copy-SI+IC*, and develops a framework to reason about replication protocols that provide SI at the global level and at the same time respect integrity constraints. The protocols developed in this thesis all offer this isolation level.

Apart of this, many existing replication protocols have some restrictions such as read-only transactions must be marked in advance [33, 34, 104, 94], or all operations of a transaction must be known upon submission time [8, 7, 90, 61]. These restrictions on applications hinder the database replication to be transparent, and make it hard to run legacy applications over the replicated infrastructure. The protocols proposed in this thesis do not have any of these restrictions and work with any database application that uses standard database interfaces.

1.4 Contribution of this thesis

In summary, this thesis makes three main contributions.

- *Correctness*: The thesis provides a complete framework to reason about SI and SI+IC in a replicated database system. Two new correctness criteria, 1-copy-SI and 1-copy-SI+IC, are proposed.
- *Practicability*: The thesis proposes a replication tool that can be used by any database application.
- *Performance*: The thesis proposes replication protocols with good performance in both LANs and WANs.

1.5 Structure of this thesis

The structure of the thesis is as follows. Chapter 2 introduces some background in regard to database replication. Chapter 3 develops a theoretical framework to reason about SI and IC in a replicated environment. Chapters 4 and 5 present the replication protocols. Chapter 4 presents protocols which guarantee 1-copy-SI and 1-copy-SI+IC. It does not consider message overhead or fault-tolerance. Chapter 5 is concerned with performance and fault-tolerance. It takes the high message delay of WANs into account and extends the protocols of Chapter 4 to work well in WANs and to be fault-tolerant. Chapter 6 presents a thorough evaluation of the protocols in LANs and WANs. Chapter 7 concludes the thesis and discusses future work.

Variations of the protocols discussed in Chapter 4 and 5 have been previously published in [73] and [75]. The protocols of this thesis vary in that they consider integrity constraints. Futhermore, part of the performance evaluation in Chapter 6 has been published in [75].

Chapter 2

Background

This chapter first shortly introduces transactions and their isolation properties since they are fundamental to replica control. A more detailed description is given in Chapter 3. Then, we give an overview of replica control principles and current solutions. Finally, we outline the goals of this thesis.

2.1 Transactions

2.1.1 Transactions and histories

A database consists of a set of data items. Database clients access the database within the boundaries of transactions. A transaction is the basic execution unit in databases [14, 120]. It contains a collection of read and write operations accessing data items within the database. If a transaction has executed successfully, the transaction *commits*. All data changes performed by committed transactions are permanent. If a transaction's execution is canceled or its results are not made permanent when it is finished, the transaction *aborts*. In this case, none of its changes will remain in the database.

In this section we use a simple notation for transactions and their execution to illustrate the basic principles. The notation is slightly different from the one used in the next chapters. We denote a transaction T_i reading data item x with value a as $r_i(x, a)$, writing value b to data item x as $w_i(x, b)$,

committing as c_i, and aborting as a_i.

For simplicity we assume that within one database the execution of operations is serial. A history represents the order of execution of transactions over time, e.g.,

$$H_{serial} : r_1(x, 0), w_1(x, 1), c_1, r_2(x, 1), w_2(y, 2), c_2$$

In history H_{serial}, all of T_1's operations happen before T_2's operations. We call this history a *serial* history, with one transaction executed completely before the other.

We say that two operations *conflict* if they are from two transactions, access the same data item, and at least one operation is a write. If one operation reads and the other writes the same data item, the corresponding two transactions have a *read/write conflict*. If both operations write the same data item, the corresponding two transactions have a *write/write conflict*. There is no conflict if two transactions read the same data item. Two transactions conflict only if they have conflicting operations. In H_{serial}, T_1 and T_2 have a read/write conflict but no write/write conflict.

2.1.2 Concurrency control and isolation levels

Note that T_1 and T_2 in H_{serial} do not interfere with each other since they execute serially. But operations of different transactions might interleave. If two transactions overlap their execution in that neither one starts after the other commits/aborts, we say that these two transactions are *concurrent* to each other. For example,

$$H_{SE} : r_1(x, 0), w_1(x, 1), r_2(x, 1), c_1, w_2(y, 2), c_2$$

T_1 and T_2 are concurrent in H_{SE}. If transactions are concurrent to each other, their execution might interfere and provide users with an incorrect image of the database and the database itself might become inconsistent.

Concurrency control is the activity of coordinating the execution of transactions that potentially interfere with each other. Concurrency control is mainly concerned with *concurrent conflicting*

transactions since transactions without conflicts will not interfere, and only concurrent transactions interleave their operations. A serial history does not allow any interference between any two transactions. However, concurrent execution allows better resource utilization and increases system throughput. Basically all database systems allow concurrent execution.

Database systems typically provide different levels of isolation that restrict the order in which a non-serial history may interleave the operations of concurrent transactions. The strongest isolation level is *serializability*, denoted as *SE* in the following. There exist various versions of serializability and we use conflict-serialiability [14]. In here, we say a history H is *serializable* if there is a serial history H_s over the same set of transactions, both histories commit the same transactions, and for every pair of conflicting operations H and H_s order them in the same way.

The most common concurrency control method to provide SE is *strict Two-Phase-Locking (2PL)* [120]. Locking requires that a transaction obtains a read (or write) lock on each data item before it reads (or writes) that data item. There can be several read locks active on the same data item (allowing concurrent reads) but when a write lock is active no other read or write lock may be granted (exclusive write access). In strict 2PL, a transaction releases all locks only at the time of commit or abort.

In recent years, a slightly weaker isolation level than SE, *Snapshot Isolation (SI)*, has been proposed [12]. Weaker means that some anomalies can occur. Nevertheless this isolation level is offered by many database systems, because the concurrency control mechanism needed to achieve this level is very efficient. A transaction executing on SI reads data from a snapshot of the committed data as of the time the transaction started. That is, if a transaction T reads data item x it reads the version of x created by a transaction T' which was the last to update x and commit before T started. If two concurrent transactions try to update the same object, one will be aborted. For example,

$$H_{SI} : r_1(x,0), w_1(x,1), r_2(x,0), c_1, w_2(x,2), a_2$$

is a history allowed under SI. In H_{SI}, assuming the original value of x is 0, T_2 reads value 0 instead of 1 even though T_1 has written value 1 to x before T_1 reads x. T_2 reads the committed version of x right before T_2 starts. T_2 needs to abort because T_1, which is concurrent to T_2 and has write/write

conflict with T_2, has committed earlier.

SI allows some non-serializable schedules. For instance,

$$H_{SI2} : r_1(x,0), r_2(y,0), w_1(y,1), w_2(x,1), c_1, c_2$$

In H_{SI2}, both transactions read from a snapshot and update different objects. The execution is SI but not SE because in a serial execution T_1 before T_2, T_2 reads $r_2(y,1)$. In a serial execution T_2 before T_1, T_1 reads $r_1(x,1)$.

With SI, we only need to worry about write/write but not read/write conflicts when determining conflicting transactions, because transactions always read committed data from a committed snapshot. The beauty of SI is that read-only transactions will never request locks, abort or interfere with update transactions. Since in database applications the number of read operations is usually much higher than that of write operations, the SI approach can have less concurrency control overhead and more concurrency compared to SE requesting locks for both reads and writes. Chapter 3 discusses SI and its implementations in detail.

In recent years, understanding isolation levels, and in particular SI, has received a lot of attention. [44, 12] provide a detailed discussion on the anomalies allowed and avoided under SI. [42] discusses how to guarantee SE if the database only provides SI. Some work also investigates weaker isolation levels than SE and SI. [34] relaxes the notion of SI by combining SI with *session* guarantees. The concept of session guarantee has been proposed in [33] and requires that a client will always see its own previous writes. [51, 13, 104, 45, 4] define a *freshness constraint* for each transaction and allow a transaction to read data satisfying its freshness constraints.

2.1.3 Integrity constraints

Data integrity is a very important characteristic of databases [120]. Database designers can specify integrity constraints and the database system guarantees that these constraints are maintained. If an update of a transaction violates a constraint, the update will typically not be executed and the transaction is aborted.

The most common integrity constraints are *unique key, primary key*, and *foreign key* constraints.

A set of attributes is considered a unique key if no two tuples in a table are allowed to have the same values in these attributes. Primary key is a special unique key which is used as the primary index of the tuples in a table (e.g., studentID in a student table). A foreign key in a table is a set of attributes that refer to the primary key of another table (e.g., the supervisorID of a student is a foreign key refering to the facultyID which is the primary key in the faculty table). The foreign key constraint refers to the requirement that the value of the foreign key attributes of a tuple must be the value of the primary key of an existing tuple in the refered table. In our example, the value of the supervisorID of a student record in the student table must be indeed the value of the facultyID of one of the faculty members.

2.2 Database replication

In this thesis we consider *full database replication.* Using *full replication*, there exist several instances of a database system and each stores a full copy of the database. An instance together with its data is also called a *database replica.* That is, if there are n replicas in the system, then there exist n physical copies for each data item x in the database. Replication is used for fault-tolerance, scalability and performance. Replication can provide fault-tolerance since if one replica fails, the other replicas can still serve client requests. Replication can be used for scalability, since client requests can be distributed across the replicas. Ideally, by adding new replicas, more client requests can be served. Alternatively, by adding new replicas, the load of each server can be reduced, and thus, the response time of individual requests can be reduced. Finally, if replicas are distributed in a wide area network, clients can access a local replica and thus, avoid wide area communication which has large message delays.

One of the challenges of replication is replica control, i.e., keeping copies consistent despite updates. Nearly all replication solutions follow the ROWA(A) approach, i.e., read-one-write-all-(available) copies approach. This means, a read operation is executed at only one replica while a write operation on a data item must update all copies of the data item that are available [1].

[1] If some replicas are currently down, then they do not need to be updated; however, once the replica comes up, it must receive the current versions of all data items.

2.2.1 Correctness criteria for replicated databases

In a replicated database, each replica executes transactions locally and produces a local history. The question now is when this distributed execution represents a globally correct execution. For example, assume two replicas A and B that execute transactions according to the ROWA approach. There are two transactions T_1 and T_2 both writing data item x. A possible execution at the both replicas could be:

$H^A : w_1(x,1), c_1, w_2(x,2), c_2$

$H^B : w_2(x,2), c_2, w_2(x,1), c_1$

Looking at the histories individually, they are both actually serial, and thus serializable. However, the serialization order is different, H^A serializes T_1 before T_2, and H^B serializes T_2 before T_1. When we look at the two database copies at the end of execution, they are not consistent. x has the value 2 in replica A, and 1 in replica B. In order to keep data consistent at all replicas, it is typically required that all replicas execute conflicting write operations in the same order. Thus, the definition of SE, or any other correctness criteria such as SI, must be extended to be meaningful in a replicated environment.

The standard correctness criterion is 1-copy-SE [14]. Despite the existence of multiple copies, a data item must appear as one logical copy (*1-copy-equivalence*). Furthermore, the execution of concurrent transactions must be coordinated such that it is equivalent to a serial execution over the logical copy (serializability). Most replica control protocols aim in providing 1-copy-SE.

Recent research has started to apply SI in a replicated database system [94, 34, 40, 73, 124, 75]. In [73], we have derived a corresponding isolation level, named 1-copy-SI. Informally, a replicated history provides 1-copy-SI if the concurrent execution of a set of transactions on the different database replicas is equivalent to executing them in a non-replicated database system under SI. 1-copy-SI is discussed in detail in Chapter 3, and the replica control protocols in this thesis provide 1-copy-SI. Concurrently to [73], Elnikety et. al. [40] have proposed a similar correctness criterion, called *generalized snapshot isolation*. However, their definitions and reasoning are quite different to [73]. [34] discusses how different degrees of session guarantees can be combined with snapshot isolation in a replicated system.

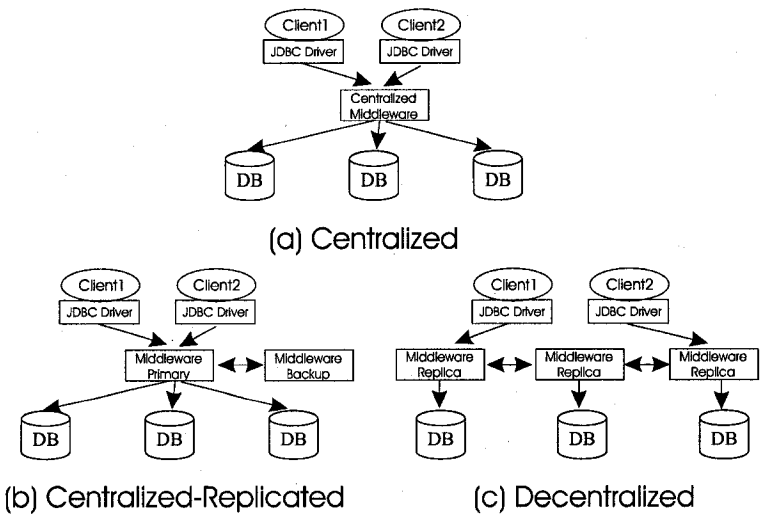


Figure 2.1: Middleware Architectures

Snapshot isolation has also been analyzed for federated database systems. In a federated system, data is distributed, not replicated, and a federation layer controls the execution of transactions across the distributed data. [109] discusses what SI means in such a federation and [108] analyzes how global SE can be maintained although the local sites only provide SI.

2.2.2 Replication architecture

Before developing replication protocols, one has to decide on the architectural framework, that is, how the replication protocols are intepreted or compiled with the existing database infrastructure.

Middleware-based replication

Recently, many *middleware-based approaches* for database replication have been proposed (e.g., [24, 61, 90, 40, 94]). A middleware approach implements replica control algorithms within a component that resides between the clients and the databases. The client only sees the middleware and sends all its requests to the middleware. The middleware then forwards these requests to individual database replicas (DB replicas) according to the replica control algorithm in order to assure the desired correctness such as 1-copy-SE or 1-copy-SI. Figure 2.1 shows three typical architectures for middleware approaches.

In the *centralized* architecture (Figure 2.1.(a)) there is only one middleware component for all databases. Obviously the middleware component is a single point of failure in such an architecture. The *centralized-replicated* architecture (Figure 2.1.(b)) improves over the centralized architecture by adding a backup middleware component. If the primary middleware fails, all clients are switched over to the backup. However, failover might be quite complicated because in case of failures the connections between the primary middleware and the DB replicas are broken. Typically, upon connection loss, database systems abort the active transactions on the connections. At the time the primary middleware crashes, a given transaction might be committed at some DB replicas, active at others, and not even started at some. The backup has to make sure that such transactions are eventually committed at all replicas. Many replicated systems follow the centralized or centralized-replicated approach [94, 7, 8, 9, 26, 62, 95, 104, 4, 45]

In the *decentralized* architecture (Figure 2.1.(c)) there is one middleware replica for each database replica. Typically the pair of middleware replica and database replica are located at one site. The middleware replicas coordinate with each other for replica control purposes. A client is connected to one middleware replica and, in case of crash of this middleware replica, is reconnected to any of the available replicas. [61, 90, 72, 73, 75, 103, 38, 89, 87, 88, 39] follow this approach.

Kernel-based replication

An alternative to a middleware-based approach would be to implement replica control within the database kernel, as an extra component that might interact with the transaction manager and the concurrency control module of the database system. [124, 67] are examples of approaches that are integrated into the kernel of an open-source database system. Basically all commercial database systems provide kernel-based replication tools. They are discussed in Section 2.2.11.

Comparison

Both middlware- and kernel-based approaches have their advantages and disadvantages. One disadvantage of kernel-based replication is that database systems are huge software systems, and integrating a completely new module into the kernel can be very complex and requires expert software

developers. Any optimization on the tightly integrated solution will be difficult to change and adjust. Furthermore, it can only be performed by the database vendor. In contrast, a middleware approach can be developed and maintained independently of the database systems, and can potentially be used in heterogeneous environments.

However, middleware-based approaches face a series of challenges. Replica control is typically tightly related with concurrency control. Since the middleware does not have full access to the concurrency control module of the database systems, it typically has to (partially) reimplement the concurrency control at the middleware level. However, only limited information is available at the middlware.

For example, the middleware does not know exactly which records are accessed by a transaction, but typically only knows which tables are accessed[2]. Hence, many middleware-based protocols (e.g., [61, 8, 7, 24]) restrict the execution of concurrent transactions if they access the same table, although they might access different records. Thus, the degree of concurrency is typically lower than with kernel-based replica control.

As another example, many database systems use locking for concurrency control purposes. As a result, a transaction T might be blocked waiting for a transaction T' to terminate and release a lock. The lock waiting queues within the kernal are typically not accessible from the middleware level. Without this blocking information, it is hard for a middleware-based protocol to discover distributed deadlocks which very possibly happen in data intensive applications.

Finally, a middleware presents an additional level of indirection and thus potentially more message overhead. This is not a problem in local area networks but can be very severe in wide area networks.

2.2.3 Categorizing replica control protocols

The seminal paper of Gray et al. [48] categorizes ROWA replica control strategies according to two parameters regarding the location of updates and the time of update propagation.

In regard to update location, a *primary copy* approach only allows data to be updated at a primary

[2]SQL statements are declaratively indicating the accessed tables while the particular records to be accessed are determined by predicates.

replica. Thus, if a client submits updates to a site other than the primary replica, the updates will be either refused or redirected to the primary site for execution. Different data items might have different primary sites. In this case, however, transactions that want to update data items with different primary sites are disallowed. In contrast, in an *update everywhere approach* the updates are accepted and executed at the local replica to which the transaction is submitted. In general, update everywhere approaches are more flexible than primary copy approaches.

Nevertheless, eventually all replicas have to perform all writes. Thus, although an update might be first executed at one replica, it must eventually be propagated to and applied at the other replicas. In *lazy replication*, updates are only propagated after commit. That is, a transaction is first executed and committed locally (at the primary in primary copy approaches or at any replica in update everywhere), and only after commit its updates are propagated to the other replicas. In contrast, using an *eager replication* approach update propagation must happen before the transaction commits, and thus, within the transaction boundaries. An eager approach provides strong data consistency because a transaction will not commit until it is certain that it will be able to commit at all other available sites. However, it delays transaction commit, and thus, increases the response time seen by the client. Transaction response time in a lazy approach is lower than that in an eager approach but it provides only weak consistency because of the early commit. A further problem of lazy approaches is that if a replica commits a transaction and then fails before propagating the updates, the other replicas will not be aware of this transaction until the replica recovers. This can severely affect data consistency.

2.2.4 Primary copy approach

In a primary copy approach, update transactions are only allowed to execute at the primary site which performs traditional concurrency control to isolate conflicting transactions. As long as other sites apply and commit updates in the same order as at the primary site, 1-copy-SE or 1-copy-SI can be provided, no matter if the changes of transactions are propagated lazily (i.e., after commit) or eagerly (i.e., before commit). However, if propagation is lazy, read-only transactions at the secondaries can read stale, i.e., outdated data. Since eager propagation delays transaction execution, most primary copy approaches are lazy [4, 13, 20, 21, 10, 94, 96, 29, 33, 34, 87, 88, 89, 95, 104].

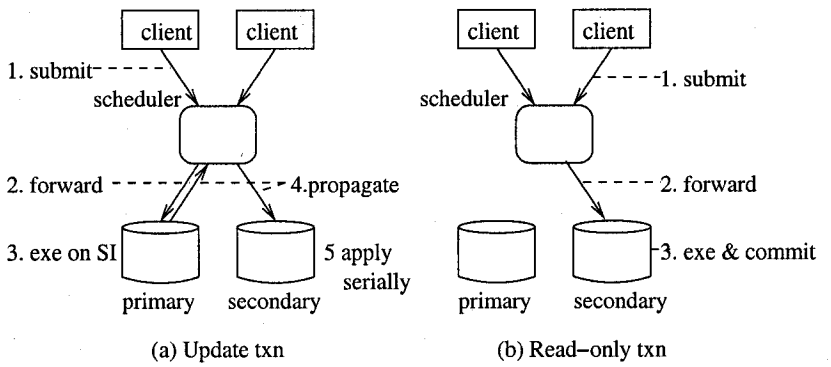


Figure 2.2: Lazy primary protocol (e.g., Ganymed [94])

Figure 2.2 shows the execution model of a typical lazy primary copy protocol (adjusted from the Ganymed [94] system). There is one central middleware (i.e., scheduler), one primary database replica, and several secondary database replicas in the system. Transactions can only be submitted to the scheduler. Figure 2.2.(a) shows how update transactions are handled. The transactions are forwarded to the primary database replica which executes the transactions using a local concurrency control mechanism to provide SE or SI locally. Then, after commit, the scheduler propagates the changes to all secondary database replicas. Secondary database replicas will apply the changes in the same order as the corresponding transactions are committed at the primary, no matter if they conflict or not. If a transaction is read-only, the scheduler forwards it to a secondary database replica for execution as shown in Figure 2.2.(b). Many lazy primary copy protocols follow this approach [4, 13, 94, 95, 96, 104].

Other lazy primary copy protocols work slightly differently. [33, 34, 88] are kernel-based and allow clients to submit requests (that represent a transaction) to local database replicas. If a transaction is read-only, it will execute locally. Otherwise, the local database replica forwards the transaction to the primary database replica which executes the transaction (using local concurrency control) and then propagates the changes to secondary database replicas after commit. The secondary database replicas work similar to Ganymed. In [34] there are additional constants on start times of update transactions at the secondaries in order that they read the same data values, and thus, execute identically to the primary.

[10] uses a global replication graph for conflict resolution. Conflict information must be sent to

one central site for building the replication graph. The communication overhead is large in WANs. [29, 20] allow multiple primaries (i.e., assigning different primaries to different data). However, if primary assignment is not done carefully, this might lead to violation of 1-copy-SE. Thus, restrictions are put on which sites can be primaries of which data items, and special update propagation paths are determined. These assignment protocols can become quite complex. Furthermore, a transaction may not update two different data items if they have different primaries.

2.2.5 Update everywhere approach

Update everywhere approaches do not require update transactions to be submitted or forwarded to a primary site for execution. However, it is more difficult to keep data consistent than in primary approaches. This is because in a primary approach conflicts between update transactions are detected in a single site (i.e., the primary) while in an update everywhere approach conflicting update transactions can run concurrently on different sites. Thus, an update everywhere approach requires additional coordination between different sites for concurrency control purposes, which is not trivial. Gray et al. [48] claim that update everywhere approaches may lead to high deadlock and abort rates if many transactions run concurrently on different sites.

Update everywhere approaches can be combined with lazy and eager propagation. The small example to illustrate the challenge of global correctness in Section 2.2.1 is using a lazy update everywhere approach. Lazy update everywhere approaches allow update transactions to be submitted and executed at any site, and then commit before their updates are propagated to other replicas. Lazy update everywhere approaches have serious data inconsistency problems. The inconsistency has to be detected and reconciled. In the example of Section 2.2.1, either the update of T_1 or T_2 has to be undone, basically rolling back an already committed transaction. Commercial systems provide a set of reconcilation strategies (e.g., let the update with the larger value always win). Saito and Shapiro [106] provide an overview of reconcilation techniques. However, these techniques are cumbersome, and lazy update everywhere is only recommended if conflict rates are extremely low.

The challenge of eager update everywhere approaches is to combine replica control with concurrency control to guarantee global transaction isolation. Traditional eager update everywhere protocols use distributed 2PL [14]. [48] has shown analytically and [67] has shown empirically

that such an approach does not scale. Recent proposals address the problems of eager update everywhere with two different approaches, either using a middleware-based scheduler or powerful communication mechanisms.

2.2.6 Update everywhere with centralized scheduler

[7] proposes a conflict aware replica control protocol, which is a typical example of an update everywhere protocol using a middleware based scheduler. There is a single scheduler in the system. It is required that all tables to be accessed in a transaction must be indicated at the start of the transaction. Upon start of a transaction, the scheduler assigns a unique version number to the transaction. Then, the scheduler requires locks for the tables to be accessed on each database replica in the order of the version numbers. Thus, all conflicting operations are enforced to execute in an identical order in all database replicas and 1-copy-SE is obtained. After being successfully scheduled, the client can submit step-by-step the read and write operations of the transaction. The scheduler forwards each write to all database replicas and returns to the client once the first database replica has executed the write. A read is sent to a single database replica. This replica must have executed all previous update operations of this transaction. Therefore, the approach is called conflict aware scheduling.

The distributed versioning protocol [8] is similar to the conflict aware protocol [7] except that [8] uses a distributed version number per table instead of a lock. C-JDBC [24] implements a table-based lock manager and uses strict 2PL. The scheduler waits until it receives responses from all database replicas involved in the operation (one for reads, all for writes) before it returns a response to the client.

2.2.7 Update everywhere based on group communication

In recent years many update everywhere protocols have been proposed [61, 90, 66, 67, 69, 103, 6, 68, 91, 90, 55, 56], that take advantage of multicast primitives provided by *group communication systems* (GCSs).

GCSs are complex software systems and have been well studied [18, 15, 28]. Examples of group communication systems include Spread [114], JGroups [49], ISIS [57], Horus [99], Ensemble [41],

Transis [37], and Totem [81]. They provide powerful primitives and have shown to be a useful abstraction for replicated and fault-toleranct systems [16]. A GCS provides multicast primitives which multicast a message to all members of a group with two semantics[3]. Namely, the GCS delivers messages in a certain *order* and with a certain *reliablility*.

The ordering semantics that are interesting in the context of database replication are *unordered*, *FIFO* (messages of one sender are received in sending order by all members), and *total* (for each two members receiving message m and m', both receive them in the same order).

The reliability semantics are *unreliable* (no guarantee that a message will be received by all members), *reliable* (whenever a member receives a message and does not fail for sufficiently long time, then all other group members will receive the message unless they fail), and *uniform reliable* (whenever a member p receives a message, all other members will receive the message unless they fail, even if p fails immediately after the reception). Additionally, choosing uniform reliable delivery, even if a member receives a message and then crashes, then all members that do not crash will not only receive the message but receive it before they are informed about the crash.

[121, 122] summarize database replication protocols using GCSs. A main categorization of protocols based on GCSs is *when multicast takes place* and *what is multicast*. We can either multicast the whole transaction before execution (see Figure 2.3.(a)) or multicast the changes performed by a transaction after execution (see Figure 2.3.(b)). Let's discuss some of the proposals in more detail.

[72, 61, 6, 90, 38] work as Figure 2.3.(a). They are middleware-based approaches. They require that all operations of a transaction must be known at start time. Read-only transactions are executed at the replica they are submitted to. A transaction request for an update transaction is multicast in *uniform reliable* and *total* order. Since *total* order guarantees transaction requests are received in the same order at all replicas, the commit order of transactions can be the same without further coordination among sites. [6, 5] apply the transactions at each site serially. In [61] transactions request all necessary locks on tables in the same order at all sites upon receiving the transaction request. [61] defines different primary sites for different transactions. A transaction is executed only at its primary after all its locks are granted, then its changes will be multicast to other sites. Other sites apply these changes again in the correct lock order.

[3] In here, we only introduce the concepts of GCS that are needed in our context.

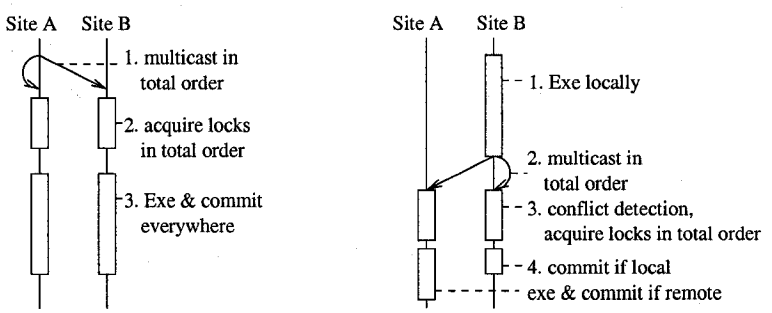


Figure 2.3: Update everywhere with GCS approach

Postgres-R [124] is kernel-based and works as Figure 2.3.(b). All databases are assured to provide SI. A transaction executes first at the replica it is submitted to. After execution, its writeset is multicast in total order to all sites for validation. If it conflicts with a concurrent transaction whose writeset was delivered earlier, it will be aborted. Otherwise, its writeset will be applied serially at remote sites according to the delivery order which is identical in all sites. Although not formally shown in [124], the approach guarantees 1-copy-SI. Postgres-R is integrated into the kernel of PostgreSQL.

GlobData [103] also performs execution before multicast, as Postgres-R. Each data item has a version number which will be increased upon a change being committed. Conflict detection is based on the version number of data accessed by transactions. In order to detect both read/write and write/write conflicts both the version numbers of data items read and the changes are multicast. Conflict detection is done upon receiving such message in total order, possibly leading to aborts. GlobData uses a GCS designed specially for WANs. The Database State Machine [92] works similarly to GlobData. However, unlike GlobData, after local execution, only identifiers of the data items read and written by update transactions and the changed values are multicast to other sites, and read-only transactions just commit locally and their read sets are not multicast. All sites apply and commit the changes serially if no conflicts are detected. Both GlobData and Database State Machine provide 1-copy-SE. [40, 39, 73, 75] also work as Figure 2.3.(b).

2.2.8 Fault-tolerance

While description above on replication approaches has focused on performance, there exist a wide range of replication solutions specifically designed for fault-tolerance. A considerable part of early research, such as [35, 46, 1], use database replication merely for fault-tolerance and/or high availability. Having more than one replica allows transactions to continue despite individual replicas not being available.

Fault-tolerance is an important topic within the distributed systems community [58, 17, 25, 50, 52, 98]. Especially in the context of group communication systems, the development of group maintenance protocols and multicast primitives with delivery guarantees has received a lot of attention [99, 81, 17, 37, 43].

Many of the replication protocols based on group communication systems, as described in Section 2.2.7, take advantage of the reliability guarantees of the GCS to provide a fault-tolerant solution. In particular, many exploit uniform reliable message delivery to guarantee that whenever a transaction commits at any replica, the transaction commits at all replicas that are currently available. Recall that uniform reliable delivery guarantees that whenever a node receives a message, all sites that are up sufficiently long will receive the message. Thus, if a replica receives a message related to a transaction and commits the corresponding transaction, uniform reliable delivery guarantees that the other available replicas will also receive the message, and thus, commit the transaction. Only replicas that have crashed might not receive the message. They have to execute and commit the transaction upon recovery. In contrast, if only reliable delivery is used, then a site might multicast a message, receive it locally, commit the corresponding transaction locally, and then crash before any other site receives the message. Thus, the other sites will not commit the transaction and the transaction is lost.

2.2.9 Load balancing

In primary copy approaches, all update transactions must be executed at the primary replica. However, read-only transactions can be executed at any replica. For update everywhere approaches, any transactions can be executed at any replica. This faciliates load balancing techniques.

Several lazy primary copy approaches attach a freshness index to secondary replicas [13, 51, 104, 45, 4] that indicate how much the secondary copies lag behind applying the updates from the primary copy. Typically, answers can be retrieved faster on staler copies. Thus, one has the possibility to trade accuracy of the returned data with the speed of receiving answers.

[95, 96] design a multi-instance database to support different applications at the same time. Each database instance is replicated using a lazy primary copy approach. There are two kinds of servers for different usage. The servers with reliable hardware support, such as Redundant Arrays of Independent Drives (RAID) [27], are used as masters, while the severs with lower reliablity guarantees are used as secondaries. All database instances share the servers. The load can be distributed among the severs for efficient resource untilization.

In the context of eager, update-everywhere protocols with a central scheduler, [9] exploits several load-balancing strategies to distribute the load over a replicated database cluster. The paper explores strategies that take query type, locality and expected execution time into account. In [26, 111], the authors explore how database replicas can join or leave a cluster in order to provide the proper amount of replicas to handle a dynamically changing workload.

2.2.10 Partial replication

So far, we have focused on full replication where each replica has copies of all data items. Using partial replication, not each data item is replicated at each site. If there are n instances of the database system, then each data item has between 1 to n data copies. The advantage is that update costs can be reduced [85]. Whenever a data item is updated not all replicas have to apply the change but only those that have a copy of the data item. Thus, more resources are available to execute further transactions. Furthermore, if data is replicated in a WAN, partial replication can significantly reduce the message overhead to keep remote replicas up-to-date.

However, partial replication faces its own challenges. Firstly, if no replica has all data items that a transaction wants to access, then either such transactions cannot be executed or transaction execution becomes distributed. This is particularly challenging if a read operation such as a complex SQL statement has to be distributed across several replicas. Furthermore, if a read operation wants to access a data item where no local copy is available, the read operation requires a remote access

resulting in high latency.

[54] proposes an epidemic protocol for partially replicated databases. The protocol aims at a WAN environment. Each data item has one or more permanent sites that always have a copy of the data item. Other sites may have a temporary cached copy. Read- and writesets are propagated to maintain consistency. If a data item is not stored at the site where the transaction executes, a request is sent to one of the permanent sites and propagated with the associated lock table information.

The aforementioned Database State Machine approach has also been extended to partial replication [112]. Partial replication has also been studied in [23]. Transactions are parsed in order to determine the sites where the transaction can be executed. This may lead to full replication when there are complex requests.

In the context of file systems and web severs, there has been considerable work on replica placement [65, 64, 70, 105, 117]. Many approaches provide heuristics of where to place data in reasonable way considering parameters such as average network latency, cache and disk capacity, and hit ratio. These approaches, however, are often not directly applicable to database replication, because requests typically only access one individual file. Thus, issues such as concurrency control, distributed queries, etc. can be ignored.

2.2.11 Commercial approaches

Basically all major database vendors provide their own replication solutions. They all prefer lazy approaches for performance reasons and primary-copy approaches for consistency reasons. They are all kernel-based approaches.

Oracle 10g Replication [100]

Oracle Enterprise Edition provides replication functionalities but its standard edition does not. Note that Oracle Enterprise Edition costs $20,000 USD per license. According to the documentation of the latest version of Oracle, v10g, Oracle provides two kinds of replication, *asynchronous* and *synchronous*.

Asynchronous replication is actually lazy replication. Oracle provides two kinds of lazy approaches, lazy primary copy and lazy update everywhere. *Snapshot replication*, also called *read-only materialized view replication*, is the lazy primary copy approach in Oracle. In snapshot replication, a snapshot (i.e., materialized view) is created at the unique primary and delivered to other sites. The materialized view is read-only but is refreshed at regular time interval.

Another option of asynchronous replication is to use updatable materialized views, also referred to as multi-master approach. This correponds to lazy update everywhere replication. An update transaction executes and commits at its local replica and then its changes are propagated to other replicas. Since lazy update everywhere may result in two concurrent transactions update the same data item at two different replicas, data reconcilation is needed. Thus, the replication tool has to detect such conflict and make sure that all replicas eventually converge to the same value for each data item. The Oracle replication management tool resolves conflicting transactions by using prebuilt conflict resolution methods. The update message contains x's old and the new value. When a replicas receives an update message with (x-old, x-new) and its current value of x is not equal to x-old, it knows that two concurrent transactions updated x. It then uses one of several existing resolution methods such as taking the maximum of the new value and the local value, or taking the value from a priority site. These methods do not guarantee data consistency in all cases. They are only applicable to specific applications and setups.

Oracle also provides synchronous replication which corresponds to eager update everywhere approaches. It uses a *Two-Phase-Commit (2PC)* to make sure all replicas commit a transaction. 2PC is expensive in terms of message overhead and latency.

Microsoft SQL Server 2005 Replication [80]

In general, Microsoft SQL Server 2005 provides replication approaches similar to the ones of Oracle. Microsoft SQL server 2005 provides three kinds of replication solutions, i.e., *snapshot replication*, *transactional replication*, and *merge replication*. The first two are for replication from server to server. The last one is for replication from server to client.

Snapshot replication in SQL server is very similar to that in Oracle. A snapshot is created at a

primary site and then propagated to secondary sites. Snapshots are usually read-only. In this case, snapshot replication is a lazy primary copy approach. SQL server also supports snapshots which can be updated occasionally. This corresponds to lazy update everywhere. A list of conflict resolution methods are provided by the replication management tool of SQL server. The methods are also the same as those in Oracle. Note that snapshot replication propagates snapshots at regular time instead of for each update transaction.

Transactional replication propagates changes on a per-transaction basis. There are three kinds of transactional replication, namely, *standard, updatable subscriber, and peer-to-peer*. Standard transactional replication is a lazy primary copy approach. Updatable subscriber transactional replication corresponds to update everywhere. It is further categorized into *immediate updating* and *queue updating*. Immediate updating uses 2PC which corresponds to eager update everywhere. Queue updating corresponds to lazy update everywhere. Conflicts are detected and resolved according to the conflict resolution policies. Peer-to-peer transactional replication is a multi-master approach which requires a careful partition of the data. It enables updates at different sites each of which is the primary of a different partition. Note that foreign key constraints and updatable primary keys are not supported in SQL server replication environments. Snapshot replication has less overhead than transactional replication because there are less data monitoring and propagation.

Merge replication is a lazy update everywhere approach using conflict resolution. It is similar to updatable snapshot replication but is designed for server to client replication.

Sybase Replication [101]

Sybase has provided replication since 1993. It supports both transactional and non-transactional replication. In regard to transaction replication, Sybase supports *One Primary - Multiple Secondary* (i.e., lazy primary copy in our definition), and *Multi-Master*. It does not support lazy update everywhere replication.

2.3 Our approach

The goal of our approach is to overcome some of the limitations of existing solutions.

- We want to have an *update everywhere approach* because primary copy approaches have some inherent limitations that cannot be avoided. Firstly, the primary copy might become a bottleneck because it has to execute all update transactions. The only primary copy approach that avoids this is one that allows different data items to have different primary copies. However, that restricts which data items a transaction might update. Secondly, the application must submit all update transactions to the primary copy, or transactions must be declared as read-only or update at start of transaction so that they can be automatically redirected to the primary or secondary copies, or transactions that are submitted to a secondary will be aborted upon submitting their first write operation. All three options are not desirable. *We want to have the option to submit any transaction to any replica in order to be transparent to the application and have the potential for load-balancing.*

- We want to have an *eager approach* because only in this way, *complex conflict resolution can be avoided and complete fault-tolerance can be achieved.* If updates are submitted only after commit and a replica fails after committing but before propagating the updates, data inconsistencies arise. In contrast, if updates are propagated before commit, all replicas are aware of such a transaction in the failure case. However, albeit being eager, *we want to keep the delay associated with eager replication as low as possible even in WAN settings.* In particular, we aim at not providing significantly worse response time than lazy approaches.

- We want the replicated system to run under snapshot isolation due to the advantages snapshot isolation has shown in non-replicated systems compared to traditional serializability. Moreover, most existing update everywhere approaches do not consider integrity constraints so that they do not work with databases with integrity constraints. Thus, *our protocols should be based on 1-copy-SI and also consider integrity constraints.*

- Our protocols should work at the *middleware level* in order to exist as an independent component and work with a heterogeneous environment. At the same time, they should avoid many disadvantages of existing middleware-based approaches. That is, they should provide *concurrency control at the record level* and not at the table level or other "coarser" concurrency levels. Furthermore, they should not pose any specific requirements at the application.

Instead, they should *work with any legacy application accessing the database through a standard interface such as JDBC [113].*

Chapter 3

Snapshot isolation and integrity constraints in a replicated system

Up to now, the most common correctness criterion for replicated databases has been 1-copy-serializability (SE) and most replica control protocols are based on 1-copy-SE. A replicated database system providing 1-copy-SE behaves as there were only one logical copy of the database providing SE.

However, snapshot isolation (SI) is becoming more and more popular since its implementation provides more concurrency as protocols implementing SE such as strict 2PL. Although SI is a weaker isolation level than SE, i.e., it allows some anomalies that are not possible under SE [12], it actually avoids all the ANSI phenomena [11]. Therefore, popular database systems such as Oracle and PostgreSQL do not provide SE at all but only run SI. Furthermore, they indicate that they provide SE according to ANSI.

Many of the replica control algorithms providing 1-copy-SE assume that the database replicas provide SE, and, e.g., use strict 2PL as concurrency control mechanism. This assumption is not valid if systems such as Oracle and PostgreSQL are used. For instance, if the database system runs SI as its highest isolation level, strict 2PL at the middleware level does not work. In this case, the system might not provide 1-copy-SE anymore. Let's look at an example with the initial values of x=0, y=0, and z=0. There are two transactions T_1 and T_2. T_1 wants to $read\ x$ and y, and then writes z to 1. T_2 wants to $read\ x$ and z, and then writes y to 2.

Assume locks are acquired subsequently at the middleware level according to strict 2PL but the underlying database provides only SI. T_1 and T_2 can start concurrently in one of the databases since they only require read locks on x at their first operations respectively. After the execution of their first operations, let's assume T_1 acquires read lock on y and write lock on z before T_2. The read operation of T_2 on z is blocked by T_1 since T_1 is holding a write lock on z. T_2 resumes after T_1 commits. However, T_2 does not read the version of z committed by T_1 since the underlying database only provides SI. Instead it reads the version committed before T_1 commits. It will lead to the following history in the underlying database.

$$H_{SI3} : r_1(x,0), r_2(x,0), r_1(y,0), w_1(z,1), c_1, r_2(z,0), w_2(y,2), c_2$$

A serial history equivalent to H_{SI3} needs to serialize T_1 before T_2 due to $r_1(y,0)$ and $w_2(y,2)$. However, $w_1(z,1)$ and $r_2(z,0)$ indicate that T_1 should be serialized after T_2. Obviously H_{SI3} is not a serializable history but a SI history. The replicated database systems does not guarantee 1-copy-SE[1].

The problems above motivate us to apply SI to a replicated environment and derive a corresponding global transaction isolation level, which we denote as 1-copy-SI. A replicated database under 1-copy-SI should behave as a non-replicated database that runs under SI. Using 1-copy-SI, we aim at achieving better performance than 1-copy-SE. Firstly, since SI allows more concurrency than SE, this increased concurrency should lead to improved throughput in the replicated case. Secondly, since SI is only concerned with write/write conflicts, we do not have any overhead in regard to reads at the replica control level. This simplifies the replication tool and can also save communication overhead compared to some 1-copy-SE protocols that require to send information about read operations for concurrency control purposes.

In this chapter, we provide a formal definition of 1-copy-SI, a correctness criterion for replicated databases. Our definition is based on the formalism introduced in [2, 3], denoted as *Generalized Isolation Definition (GID)*, to reason about Snapshot Isolation. We then give some necessary and

[1]To achieve 1-copy-SE, the acquisition of locks at middleware levels must be atomic which is not required by strict 2PL. Actually many middleware based protocols follow this strategy such as [7, 61, 90].

sufficient conditions for a replicated history to be 1-copy-SI. Then, we extend the formalism to be able to express integrity constraints (IC) in a way that conforms with how they are handled in existing commercial systems that run under SI. From there, we derive an extended correctness criterion, denoted as 1-copy-SI+IC which provides snapshot isolation and proper handling of integrity constraints in a replicated environment. Again, we identify conditions that allow us to determine whether a given replicated history is 1-copy-SI+IC.

3.1 Snapshot Isolation (SI)

[12] gives a first, rather informal description of SI. SI is defined by two properties. The first property, referred to as *Snapshot-Read* property by [2], indicates that a transaction T on SI reads data from a snapshot which contains all updates committed before T starts (plus its own updates). The second property, referred to as *Snapshot-Write* property by [2], indicates that no two concurrent transaction may write the same object. That is, if two concurrent transactions both want to write the same data item only one of them will be allowed to commit. Snapshot isolation avoids the ANSI anomalies [11] but is not serializable in the strict sense. We gave an example in Section 2.1.2.

Despite not being serializable, SI is attractive because it generally allows for more concurrency than strict 2PL. Thus, database systems such as Oracle, Microsoft SQL Server, PostgreSQL now support it. Commercial systems usually implement Snapshot-Read via a multi-version system: a write of transaction T on data item x creates a new version, a read of transaction T on data item x reads the last version of x that was committed before T started (or its own version if it has created one). Thus, reads do not set any locks. Snapshot-Write is typically implemented by letting transactions set long exclusive locks on data items they want to write. When T receives the lock on x, T checks the latest committed version of x. If it was created by a transaction T' concurrent to T (i.e., T' committed after T started), then T aborts, otherwise it performs the update and continues execution. This is often denoted as *first-updater-wins* technique. Alternatively, transactions could perform their updates optimistically and only check at commit time whether a concurrent transaction that already committed had conflicting updates. This technique is denoted as *first-committer-wins* strategy.

Having no locks for reads can increase concurrency significantly. However, aborting one of two concurrent transactions having conflicting writes can lead to higher abort rates than standard strict 2-phase-locking.

Our correctness reasoning is based on the formalism introduced in [2, 3], denoted as GID. In his thesis [2], Adya defines GID and uses it to reason about various isolation levels in a non-replicated environment, including snapshot isolation. GID is a very powerful tool and allows reasoning about correctness that is independent of the actual implementation. In the remainder of this section, we present GID for snapshot isolation. We only slightly modify the notation to adjust it better to our needs.

3.1.1 Transactions and histories in General Isolation Definition (GID)

A data item x (also referred to as object) of the database has a life time from its initial unborn version, x_{init}, to its dead version, x_{dead} created by a transaction deleting x. A transaction T_i starts with a start operation s_i, then contains a sequence of read and write operations, and terminates with a commit operation (i.e., c_i) or an abort operation (i.e., a_i). A transaction T_i creates a version x_i of object x by performing a write operation $w_i(x_i)$. If T_i reads x it reads a specific version x_j, denoted as $r_i(x_j)$. If T_i writes x, then it $installs\ x_i$ when it commits. For simplicity, we assume T_i will not read or write the same object twice, and if it reads and writes an object, it performs the read before the write.

Let $\mathcal{T}$ be a set of transactions. A history H over $\mathcal{T}$ describes the execution of the transactions in $\mathcal{T}$ and consists of two parts. Firstly, it has a partial order[2], called time-precedes order $\prec_t$, over all operations of transactions of $\mathcal{T}$ with the following properties:

1. It includes the order in which operations within a transaction are executed. That is, for any two operations o_{ij} and o_{ik} of $T_i \in \mathcal{T}$, if o_{ij} happens before o_{ik} in the execution, then $o_{ij} \prec_t o_{ik}$. In particular $s_i \prec_t c_i$.

2. If $w_i(x_i)$ and $r_j(x_i)$, then $w_i(x_i) \prec_t r_j(x_i)$.

[2]Partial order in this thesis refers to an order $<$ with irreflexivity (i.e., $\neg(a < a)$) and transitivity (i.e., $(a < b) \wedge (b < c) \Rightarrow (a < c)$.

3. For any two committed transactions T_i and T_j: either $c_i \prec_t s_j$ or $s_j \prec_t c_i$.

Secondly, H provides a version order, $\ll$, that is a total order on the versions of each committed object. For any object x, x_{init} is always considered the smallest object version.

For convenience, we will present a history H as a sequence of operations (i.e., start, read, write, commit, abort) with a total order (from left to right) consistent with $\prec_t$. Furthermore, we omit that x_{init} is smaller than any other version of x. For example, consider the history $H_{write-order}$:

$H_{write-order}$: $s_1, s_2, w_1(x_1), w_2(x_2), w_2(y_2), c_1, c_2, s_3, r_3(x_1), c_3, s_4, w_4(y_4), a_4$ $[x_2 \ll x_1, y_2]$

This history shows how little restrictions are actually in place. In here, x_2 is ordered before x_1 in the version order, although in $\prec_t$, $w_1(x_1)$ is ordered before $w_2(x_2)$, and also the commit order is c_1 before c_2. Furthermore, T_3 reads x_1 although x_2 was created later. Furthermore, y_4 is not considered in the version order since it was created by an aborted transaction.

In the following, our example histories often do not start with an empty database but assume that before the history H over a set of transaction $\mathcal{T}$ started, there executed transactions, e.g., transaction T_0, that committed before H started. If T_0 wrote object version x_0, then we assume that $x_0 \ll x_i$ for any transaction T_i in $\mathcal{T}$ that writes x during H.

3.1.2 Snapshot isolation in GID

[2] now formally defines Snapshot-Read and Snapshot-Write as follows:

Definition 1. ***Snapshot-Read.*** *All read operations performed by a transaction T_i occur at its start point. That is, if $r_i(x_j)$ occurs in history H, then:*

1. $c_j \prec_t s_i$, *and*
2. *if $w_k(x_k)$ also occurs in $H (j \neq k)$, then either*
   - *(a)* $s_i \prec_t c_k$, *or*
   - *(b)* $c_k \prec_t s_i$ *and* $x_k \ll x_j$

Definition 2. ***Snapshot-Write.*** *If T_i and T_j are concurrent and both commit, they can not both modify the same object. That is, if $w_i(x_i)$ and $w_j(x_j)$ both occur in history H, then either $c_i \prec_t s_j$ or $c_j \prec_t s_i$.*

GID makes use of data-flow graphs to reason about the properties of a history. In the context of SI, it introduces the notion of a *Start-ordered Serialization Graph (SSG)*, that records the dependencies between transactions for a given history H over $\mathcal{T}$. In the following, we say T_j *directly write-depends* on T_i if both write a common data item x and x_i and x_j are consecutive versions of x in H's version order. T_j *directly read-depends* on T_i if it reads a version of an object created by T_i. T_j *directly anti-depends* on T_i if T_i reads a version of an object x and T_j creates x's next version in the version order. T_j *start-depends* on T_i if T_i commits before T_j starts in the time-precedes order. The dependency definitions are summarized in Table 3.1[3].

Definition 3. ***Start-ordered Serialization Graph (SSG).*** *The $SSG(H)$ of a history H over a set of transaction $\mathcal{T}$ is a directed graph where each node in $SSG(H)$ corresponds to a committed transaction in H, and there is a write-, read-, anti-, or start-dependency edge from T_i to T_j iff T_j directly write-, directly read-, directly anti-, or start-depends on T_i, respectively.*

In the following, given the $SSG(H)$ of a history H, we denote as $T_i \xrightarrow{ww^+} T_j$ a path in the graph from T_i to T_j consisting only of write-dependency edges. Similarily, we denote as $T_i \xrightarrow{S^+} T_j$ a path in $SSG(H)$ with only start-dependency edges.
From there, GID identifies five phenomena that a history must avoid to be SI.

- **G-1a: Aborted Reads.** A history H over $\mathcal{T}$ exhibits phenomenon G-1a if it contains an aborted transaction T_1 and a committed transaction T_2 such that T_2 has read some objects modified by T_1.

- **G-1b: Intermediate Reads.** A history H exhibits phenomenon G-1b if it contains a committed transaction T_2 that has read a version of object x written by transaction T_1 that was not T_1's final modification of x. We do not further consider this phenomena because our transaction model assumes that each transaction only writes an object at most once.

[3]GID also considers predicate read and write operations. For simplicity, we do not discuss them but we believe our definitions and theorems can be easily extended to accomodate them.

Dependency Type	Description	SSG	Edge name
Directly write-depends	T_i installs x_i and T_j installs x's next version	$T_i \xrightarrow{ww} T_j$	write-dependency edge
Directly read-depends	T_i installs x_i and x_i is the same version of x in T_j's read	$T_i \xrightarrow{wr} T_j$	read-dependency edge
Directly anti-depends	T_i reads x and T_j installs x's next version	$T_i \overset{rw}{-\!\!\rightarrow} T_j$	anti-dependency edge
start-depends	T_j starts after T_i commits	$T_i \xrightarrow{S} T_j$	start-dependency edge

Table 3.1: Dependencies (based on Fig. 2 in [3])

- **G-1c: Circular Information Flow.** A history H exhibits phenomenon G-1c if the direct serialization graph $SSG(H)$ contains a directed cycle consisting entirely of write-dependency and read-dependency edges. We refer to such cycle as a *G-1c cycle.*

- **G-SIa: Interference.** A history H exhibits phenomenon G-SIa if $SSG(H)$ contains a read- or write-dependency edge from T_i to T_j without there also being a start-dependency edge from T_i to T_j.

- **G-SIb: Missed Effects.** A history H exhibits phenomenon G-SIb if $SSG(H)$ contains a directed cycle with exactly one anti-dependency edge. We refer to such cycle as a *G-SIb cycle.*

GID defines an isolation level PL-SI corresponding to SI as the one in which the G1a, G1b, G1c, G-SIa, and G-SIb phenomena are disallowed. Roughly, G1a-c capture the essence of *dirty read* and *dirty write* while G-SIa-b capture the essence of violating *Snapshot-Read* and *Snapshot-Write*.[4] For the convenience of discussion, we refer to a history as a *SI-history* if it avoids phenomena G-1 and G-SI.

Example 1. *H_{non-SI} is not a SI-history while H_{SI} is a SI-history. Their SSGs are shown in Figure 3.1 and 3.2 respectively. We assume that a transaction T_0 installs version x_0 and y_0 before the transactions T_1 to T_3 start.*

[4]We refer to [2, 3] for the proofs that G1 and G-SI are necessary and sufficient conditions for a history to provide Snapshot Read and Snapshot Write.

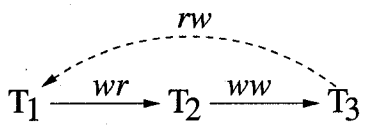


Figure 3.1: SSG(H_{non-SI}) in Example 1

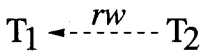


Figure 3.2: SSG(H_{SI}) in Example 1

H_{non-SI}: $s_1, s_2, s_3, r_3(x_0), w_1(x_1), c_1, r_2(x_1), w_2(y_2), c_2, w_3(y_3), c_3 \quad [x_1, y_2 \ll y_3]$

H_{SI}: $\quad s_1, s_2, s_3, r_3(x_0), w_1(x_1), c_1, r_2(x_0), w_2(y_2), c_2, w_3(y_3), a_3 \quad [x_1, y_2]$

In both H_{non-SI} and H_{SI}, T_1 installs the version x_1 following x_0. In H_{non-SI}, T_2 reads the version of x created by T_1 ($r_2(x_1)$). This violates Snapshot-Read because T_1 has not committed at the time T_2 starts. Correspondingly we can see that there is a $T_1 \xrightarrow{wr} T_2$ edge but no $T_1 \xrightarrow{S} T_2$ edge in $SSG(H_{non-SI})$ (Figure 3.1). This means H_{non-SI} has phenomenon G-SIa. Moreover, T_2 and T_3 both write y concurrently and both are allowed to commit. This violates Snapshot-Write. Correspondingly we can see that there is a $T_2 \xrightarrow{ww} T_3$ edge but no $T_2 \xrightarrow{S} T_3$ edge in $SSG(H_{non-SI})$. Furthermore, there is a G-SIb cycle $T_1 \xrightarrow{wr} T_2 \xrightarrow{ww} T_3 \xrightarrow{rw} T_1$ in $SSG(H_{non-SI})$ having exactly one anti-dependency edge. Thus, H_{non-SI} also has phenomenon G-SIb.

In H_{SI}, T_2 reads x from T_0 instead of T_1 ($r_2(x_0)$). This is correct, because T_2 started after T_0 committed. Although T_1 and T_2 are concurrent, both are able to commit because they write different objects. However, T_3 is aborted because it writes y, is concurrent to T_2, and T_2 commits (only one may commit). Figure 3.2 shows SSG(H_{SI}). It is easy to verify that H_{SI} avoids phenomena G-1a, G-1b, and G-SIa. Since SSH(H_{SI}) is acyclic, G-1c and G-SIb are avoided. Hence, H_{SI} is a SI-history.

3.1.3 Observations

Here we discuss some further observations and properties of SI-histories and general histories and their SSGs. They will be useful when we discuss SI in a replicated system.

First of all, we want to point out a property that holds in the $SSG(H)$ of any history H. Figure 3.3 shows an illustration of this property.

Proposition 1. *Let H be a history over $\mathcal{T}$. Let $T_i, T_j \in \mathcal{T}$ be two transactions writing x, and*

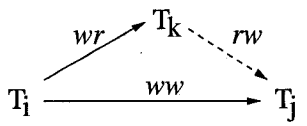


Figure 3.3: Relationship of read-, write-, and anti-dependency edge

Dependency	Order Requirement in SI-history
$T_i \xrightarrow{S} T_j$	$c_i \prec_t s_j$
$T_i \xrightarrow{ww} T_j$	$c_i \prec_t s_j$
$T_i \xrightarrow{wr} T_j$	$c_i \prec_t s_j$
$T_i \overset{rw}{-\!\!\rightarrow} T_j$	$s_i \prec_t c_j$

Figure 3.4: Order requirements for SI-histories

$T_k \in \mathcal{T}$ be a transaction reading x. If $T_i \xrightarrow{ww} T_j$ and $T_i \xrightarrow{wr} T_k$ are two edges in $SSG(H)$, then $T_k \overset{rw}{-\!-\!\rightarrow} T_j$ appears in $SSG(H)$.

Proof. $T_i \xrightarrow{ww} T_j$ means that x_i and x_j are consecutive versions in x's version order. $T_i \xrightarrow{wr} T_k$ means that T_k reads version x_i. Since T_j installs the next version of x_i, according to the construction of direct anti-dependency edges, there must be a $T_k \overset{rw}{-\!\rightarrow} T_j$ edge in $SSG(H)$. □

Secondly, we want to look at the relationship between dependencies and the start and commit order of transactions. We have shown with our first example of a history, $H_{write-order}$, that there are generally very little restrictions of how operations are $\prec_t$-ordered in a history. However, a SI-history has quite strong properties in regard to the $\prec_t$-order. Every dependency edge in the $SSG(H)$ of a SI-history H indicates some ordering between the start and commit operations of the involved transactions. Table 3.4 indicates these ordering implications. Clearly, a start-dependency edge between T_i and T_j means $c_i \prec_t s_j$ for any history H by definition. Furthermore, in order to avoid G-SIa, the $SSG(H)$ of a SI-history H must have a start-dependency edge whenever there is a write- or read-dependency edge. This means, whenever there is a write- or a read-dependency edge from T_i to T_j, we have $c_i \prec_t s_j$ in H. Finally, an anti-dependency $T_i \overset{rw}{-\!\rightarrow} T_j$ implies $s_i \prec_t c_j$ in H. Assume that this would not be the case. Then $c_j \prec_t s_i$ holds. Thus, there would be a start-dependency edge $T_j \xrightarrow{S} T_i$ resulting in a cycle between T_i and T_j with exactly one anti-dependency edge. This is phenomenon G-SIb and avoided by SI-histories.

Finally, we can observe that in a SI-history the version order and $\prec_t$-order of commit operations are highly related.

Lemma 1. *Let H be a SI-history over $\mathcal{T}$ and let $T_i, T_j \in \mathcal{T}$ be two transactions writing x. $x_i \ll x_j$ iff $c_i \prec_t c_j$.*

Proof. First, one has to note that when an object version x_i appears in the version order, T_i must have committed according to the definition of a history.

1. $x_i \ll x_j \Longrightarrow c_i \prec_t c_j$

   By the definition of write-dependency edges (as in Table 3.1), if $x_i \ll x_j$, then $SSG(H)$ has a path $T_i \xrightarrow{ww^+} T_j$ consisting only of write-dependency edges . Since H is a SI history, it avoids G-SIa, and each write-dependency edge is accompanied with a start dependency edge. Thus, we derive that $SSG(H)$ also contains $T_i \xrightarrow{S^+} T_j$. This, together with $s_i \prec_t c_i$ results in $s_i \prec_t c_i \prec_t ... \prec_t s_j \prec_t c_j$. Hence, $c_i \prec_t c_j$.

2. $c_i \prec_t c_j \Longrightarrow x_i \ll x_j$

   Assume $x_j \ll x_i$. Based on the first part of the proof above, $c_j \prec_t c_i$. Hence, $c_j \prec_t c_i \prec_t c_j$ which is impossible since $\prec_t$ is irreflexive. Thus, $x_i \ll x_j$ must hold.

□

3.2 Snapshot isolation in a replicated system

In this section we extend the notion of SI to a replicated environment. In order for a replicated database to provide a certain level of isolation, it should behave like a non-replicated database that runs under this isolation level. The concept of 1-copy-SE is well known and understood ([14]). It requires the execution in the replicated system to be equivalent to a serial execution in a non-replicated system. In this section, we formally define what it means for a history to be 1-copy-snapshot-isolation (1-copy-SI), and discuss necessary and sufficient conditions for a history to be 1-copy-SI.

3.2.1 Transactions and histories in a replicated database

A replicated database consists of a set of replicas $\mathcal{R}$ each of which keeps a copy of the database. Our model follows a Read-One-Write-All (ROWA) approach in which each update transaction has one local replica that performs all its operations. The transaction is called local at this replica, and remote at the other replicas. Only the write operations of a transaction are applied at the remote replicas. Hence, all replicas execute the same set of update transactions, but an update transaction T_i has a readset RS_i consisting of all read operations only at one replica while it has the same writeset WS_i consisting of its write operations at all replicas. Read-only transactions, in contrast, only exist at the local replica. We express this by using a ROWA mapper function.

Definition 4. ***Mapper function.*** *A ROWA mapper function, $rmap$, takes a set of transactions $\mathcal{T}$ and a set of replicas $\mathcal{R}$ as input, and transforms $\mathcal{T}$ into a set of transactions $\mathcal{T}' = rmap(\mathcal{T}, \mathcal{R})$. $rmap(\mathcal{T}, \mathcal{R})$ transforms each update transaction $T_i \in \mathcal{T}$ into a set of transactions $\{T_i^k | R^k \in \mathcal{R}\}$. In this set there is exactly one local transaction T_i^l where $WS_i^l = WS_i$ and $RS_i^l = RS_i$ (T_i is local at R^l). The rest are remote transactions T_i^r, where $WS_i^r = WS_i$ and $RS_i^i = \emptyset$ (T_i is remote at R^r). A read-only transaction T_i is transformed into a single local transaction T_i^l with $RS_i^l = RS_i$. We denote as $\mathcal{T}^k = \{T_i^k | T_i^k \in \mathcal{T}'\}$ the set of transactions executed at replica R^k.*

Executing $\mathcal{T}'$ at the replicas $\mathcal{R}$ leads to what we denote a replicated history.

Definition 5. ***Replicated history.*** *Let $\mathcal{T}$ be a set of transactions, $\mathcal{R}$ a set of replicas and $rmap$ a ROWA mapper function generating $\mathcal{T}' = rmap(\mathcal{T}, \mathcal{R})$. Let RH^k be a local history over $\mathcal{T}^k$ at $R^k \in \mathcal{R}$. We denote the union over all local histories RH^k as a* replicated RH over $rmap(\mathcal{T}, \mathcal{R})$, *i.e., $RH = \bigcup RH^k, R^k \in \mathcal{R}$.*

3.2.2 1-copy-SI

We now have to define when a replicated history provides 1-copy-SI, i.e., when it is equivalent to a SI-history over a non-replicated database. We model this by requiring a replicated history over $\mathcal{T}$=rmap($\mathcal{T}$, $\mathcal{R}$) to have the same dependencies between read and write operations as a non-replicated SI-history over $\mathcal{T}$. In GID, any such dependency is captured by the means of a write-, read- or anti-dependency edge in the SSG. A replicated RH is the union of the subhistories RH^k at the different replicas. Each RH^k has its own $SSG(RH^k)$ reflecting the dependencies that occurred in this history. The union of all these $SSGs$ reflects the sum of all dependencies. Thus, an equivalent, non-replicated SI-history has to have the same dependencies. We first define these set of dependencies as a graph:

Definition 6. ***Union Serialization Graph (USG).*** *Let $RH = \bigcup RH^k$ be a replicated history over $rmap(\mathcal{T}, \mathcal{R})$. We denote as $USG(RH)$ the following graph.*

1. *For each $R^k \in \mathcal{R}$, if $SSG(RH^k)$ has node $T_i^k \in \mathcal{T}^k$, then $USG(RH)$ has a node T_i.*
2. *For each $R^k \in \mathcal{R}$ and each write-, read-, or anti-dependency edge from T_i^k to T_j^k in $SSG(RH^k)$, $USG(RH)$ has a corresponding write-, read-, or anti-dependency from T_i to T_j.*
3. *There are no further edges or nodes in $USG(RH)$.*

Definition 7. ***1-copy-SI.*** *Let $RH = \bigcup RH^k$ be a replicated history over $rmap(\mathcal{T}, \mathcal{R})$. We say RH is 1-copy-SI if*

1. *For each $R^k \in \mathcal{R}$, RH^k is a SI-history.*
2. *For all update transactions $T_i \in \mathcal{T}$ and for all $R^k, R^l \in \mathcal{R} : c_i^k \iff c_i^l$.*

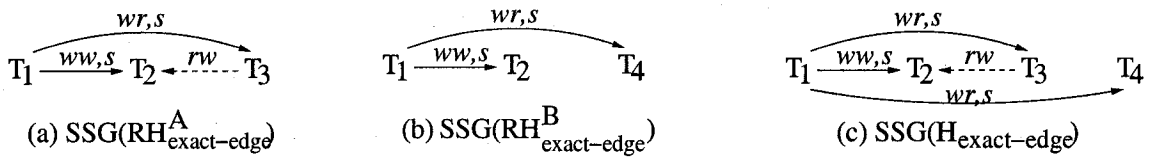


Figure 3.5: SSGs of Example 2

3. *There exists a SI-history H over $\mathcal{T}$ such that,*

   (a) $SSG(H)$ and $USG(RH)$ have the same nodes;

   (b) $SSG(H)$ has exactly the same write-, read-, and anti-dependency edges as $USG(RH)$.

(1) means that the histories at all replicas must be SI-histories. In the following we often refer to them as the *local histories*. (2) means all local histories must commit the same set of update transactions. Finally, (3) means a SI-history over the original set of transactions must exist with the same dependencies. We refer to this non-replicated history over $\mathcal{T}$ often as a *global* history.

Example 2. *In this example, there are two replicas R^A and R^B. Transactions T_1, T_2, and T_3 are local at R^A while T_4 is local at R^B. The replicated history $RH_{exact-edge}$ is the union of the local histories $RH^A_{exact-edge}$ and $RH^B_{exact-edge}$*

$$RH^A_{exact-edge} : s^A_1, w^A_1(x_1), w^A_1(y_1), c^A_1, s^A_2, w^A_2(x_2), s^A_3, c^A_2, r^A_3(x_1), c^A_3 \; [x_1 \ll x_2, y_1]$$
$$RH^B_{exact-edge} : s^B_1, w^B_1(x_1), w^B_1(y_1), c^B_1, s^B_2, w^B_2(x_2), s^B_4, c^B_2, r^B_4(y_1), c^B_4 \; [x_1 \ll x_2, y_1]$$

$SSG(RH^A_{exact-edge})$ and $SSG(RH^B_{exact-edge})$ are shown in Fig. 3.5. For simplicity, the superscript A and B at the transactions are omitted. It is easy to verify that both RH^A and RH^B are SI-histories. $USG(RH)$ is the union graph of all write-, read- and anti-dependency edges of $SSG(RH^A_{exact-edge})$ and $SSG(RH^B_{exact-edge})$.

We can show that the replicated history $RH_{exact-edge}$ is 1-copy-SI by building the following global history $H_{exact-edge}$ over $\{T_1, T_2, T_3, T_4\}$:

$$H_{exact-edge}: \quad s_1, w_1(x_1), w_1(y_1), c_1, s_2, w_2(x_2), s_3, s_4, c_2, r_3(x_1), c_3, r_4(y_1), c_4 \; [x_1 \ll x_2, y_1]$$

$SSG(H_{exact-edge})$ is shown in Fig. 3.5.(c). It has exactly the same write-, read- and anti-dependency edges as $USG(RH_{exact-edge})$. We can also easily see that H avoids G1 and G-SI. Hence, $RH_{exact-edge}$ is 1-copy-SI.

In above example, we have shown that $RH_{exact-edge}$ is 1-copy-SI by constructing a non-replicated history $H_{exact-edge}$ that fulfills the conditions of the 1-copy-SI definition. However, constructing an appropriate non-replicated global SI-history for an arbitrary replicated history that fulfills the 1-copy-SI property is not always trivial. Furthermore, in case a replicated history is not 1-copy-SI, it is difficult to prove that no global SI-history with the appropriate properties exists. Thus, we need a more convenient way to determine whether a replicated history is 1-copy-SI.

For 1-copy-serializability and ROWA, [14] simply checked whether the union of the serialization graphs of the histories at the different replicas are acyclic. The question arises, whether we can simply check $USG(RH)$ to determine whether the execution is 1-copy-SI or not.

3.2.3 Necessary conditions for a replicated history to be 1-copy-SI

It is clear that if $USG(RH)$ has a G-1c or G-SIb cycle, then RH cannot be 1-copy-SI because it is not possible for a SI-history H to have a $SSG(H)$ with the same edges. Our first question is whether any other characteristics of $USG(RH)$ can be determined that make it clear that RH is not 1-copy-SI. Let's have a look at an example.

Example 3. *In this example, there are two replicas R^A and R^B. Transaction T_1 and T_2 are local at R^A, T_3 and T_4 are local at R^B. We assume an initial transaction T_0 created x_0 and y_0 and committed before the following execution starts.*

$$RH^A_{hole} : s^A_1, w^A_1(x_1), c^A_1, s^A_2, r^A_2(x_1), r^A_2(y_0), c^A_2, s^A_4, w^A_4(y_4), c^A_4\ [x_1, y_4]$$
$$RH^B_{hole} : s^B_4, w^B_4(y_4), c^B_4, s^B_3, r^B_3(y_4), r^B_3(x_0), c^B_3, s^B_1, w^B_1(x_1), c^B_1\ [x_1, y_4]$$

SSG(RH^A_{hole}) and SSG(RH^B_{hole}) are shown in Figures 3.6.(a) and (b) respectively. The $USG(RH)$ shown in Figure 3.6 (c) has no G-1c or G-SIb cycles. Still, RH_{hole} is not 1-copy-SI.

We show by contradiction that RH_{hole} is not 1-copy-SI. Assume RH_{hole} is 1-copy-SI. Then

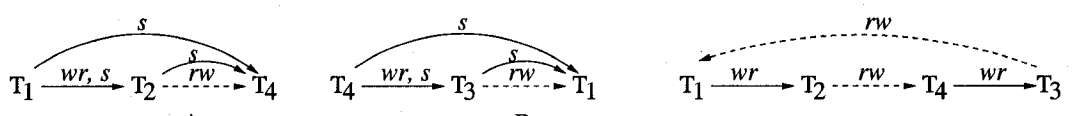


Figure 3.6: SSGs of Example 3

there must be a global SI-history H_{hole} which contains the same write-, read-, and anti-dependency edges as $USG(RH_{hole})$. Hence, based on $T_1 \xrightarrow{wr} T_2 \overset{rw}{-\!\!\rightarrow} T_4$ in $USG(RH_{hole})$ and Table 3.4, we derive for the $\prec_t$-order of H:

$$\left.\begin{array}{l} T_1 \xrightarrow{wr} T_2 \Longrightarrow c_1 \prec_t s_2 \\ T_2 \overset{rw}{-\!\!\rightarrow} T_4 \Longrightarrow s_2 \prec_t c_4 \end{array}\right\} \Longrightarrow c_1 \prec_t c_4$$

Similarily, due to $T_4 \xrightarrow{wr} T_3 \overset{rw}{-\!\!\rightarrow} T_1$ we derive:

$$\left.\begin{array}{l} T_4 \xrightarrow{wr} T_3 \Longrightarrow c_4 \prec_t s_3 \\ T_3 \overset{rw}{-\!\!\rightarrow} T_1 \Longrightarrow s_3 \prec_t c_1 \end{array}\right\} \Longrightarrow c_4 \prec_t c_1$$

This results in $c_1 \prec_t c_4 \prec_t c_1$ which is impossible since $\prec_t$ is irreflexive. Thus, no SI-history could have a graph with above edges, and RH_{hole} is not 1-copy-SI.

The problem of RH_{hole} is that a global SI-history can simply not behave in the same way. T_1 and T_4 do not conflict. So their order does not seem to matter. However, T_2 reads x and y from a snapshot after T_1 commits but before T_4 commits in R^A. This indirectly requires T_1 to commit before T_4. In contrast, T_3 reads x and y from a snapshot after T_4 commits but before T_1 commits in R^k, indirectly ordering T_4 before T_1. In a non-replicated history, only one of the snapshots is possible, that is either T_1 commits before T_4 or it commits after T_4 but not both.

The problem is that $USG(RH_{hole})$ (see Figure 3.6.(c)) has a cycle with more than one anti-dependency edge. In principle, this is allowed by the definition of SI. But it turns out that the particular cycle above is not possible in a non-replicated history.

Thus, we define a further phenomenon.

- **G-SIb*: Anti-dependency cycle** A history H exhibits phenomenon G-SIb* if $SSG(H)$ has

a cycle with at least one anti-dependency edge and each anti-dependency edge is prefixed by a write-, read-, or start-dependency edge. We refer to such a cycle as a G-SIb* cycle.

G-SIb* refers to cycles where there are no consecutive anti-dependency edges[5]. Note that G-SIb* actually includes G-SIb because if there is a cycle with exactly one anti-dependency edge, then this anti-dependency edge must be prefixed with a non anti-dependency edge, i.e., a write-, read-, or start-dependency edge. G-SIb* is a derived phenomenon, i.e., if a history avoids G-1a-c and G-SIa-b, then it automatically avoids G-SIb*.

Lemma 2. *A (non-replicated) SI-history H over a set of transactions $\mathcal{T}$ avoids G-SIb**

Proof. Assume there is a SI-history H that has phenomenon G-SIb*. $SSG(H)$ cannot have a cycle with only one anti-dependency because it avoids G-SIb. Thus, $SSG(H)$ has a cycle c with $m\,(m > 1)$ anti-dependency edges and each anti-dependency edge is prefixed by a write-, read-, or start-dependency edge. Firstly, we can easily derive that $SSG(H)$ must have a cycle c' with $m\,(m > 1)$ anti-dependency edges and all other edges in the cycle are start-dependency edges. This is because whenever there is a write- or read-dependency edge between from T_i to T_j there is also a start-dependency edge because of G-SIa. Thus, in the following, we only consider a cycle that consists of m anti-dependency edges, all other edges are start-dependency edges, and each anti-dependency edge is prefixed by a start-dependency edge. We can break the cycle into m sections. Each section $k \in \{0, \ldots, m-1\}$ has the pattern $T_{i_k} \xrightarrow{s^+} T_{j_k} \xrightarrow{rw} T_{i_{(k+1)\%m}}$. According to Table 3.4, we can derive for the $\prec_t$-order of H for each section k due to transitivity:

$$\left.\begin{array}{l} T_{i_k} \xrightarrow{s^+} T_{j_k} \Rightarrow c_{i_k} \prec_t s_{j_k} \\ T_{j_k} \xrightarrow{rw} T_{i_{(k+1)\%m}} \Rightarrow s_{j_k} \prec_t c_{i_{(k+1)\%m}} \end{array}\right\} \Rightarrow c_{i_k} \prec_t c_{i_{(k+1)\%m}}$$

If we now look at all sections, we obtain: $c_{i_0} \prec_t c_{i_1} \prec_t \cdots \prec_t c_{i_k} \prec_t c_{i_{(k+1)}} \cdots \prec_t c_{i_m-1} \prec_t c_{i_0}$. Since $\prec_t$ is irreflexive this results in a contradiction. □

Thus, coming back to Example 3, since $USG(RH_{hole})$ has a G-SIb* cycle, we can immediately see that RH is not 1-copy-SI because there cannot be a global SI-history with the same dependency edges (and cycle).

[5]SI allows cycles with two consecutive anti-dependency edges. Interestingly, [42] determines that histories that are SI but not SE are exactly those that contain cycles with consecutive anti-dependency edges.

In summary we observe the following necessary conditions to be 1-copy-SI: if a replicated history RH is 1-copy-SI, then $USG(RH)$ has no G-1c or G-SIb* cycles.

3.2.4 Sufficient conditions for a replicated history to be 1-copy-SI

It turns out that avoiding G-1c and G-SIb* is not only necessary but also sufficient for a replicated RH history to be 1-copy-SI. That is, for a replicated history RH, if all local histories RH^k are SI, all R^k commit the same update transactions, and $USG(RH)$ has no G-1c and G-SIb* cycles, then RH is 1-copy-SI. In particular, we are able to construct a global SI-history H such that $SSG(H)$ has the same write,- read- and anti-dependency edges as $USG(RH)$. We start with some interesting properties of a RH whose local histories are SI-histories.

Lemma 3. *Let RH be a replicated history over $rmap(\mathcal{T}, \mathcal{R})$. At each $R^k \in \mathcal{R}$, let RH^k be a SI-history over $\mathcal{T}^k$. Let each update transaction $T_i \in \mathcal{T}$ commit at either all or none of the replicas.*

If $USG(RH)$ has no G-1c cycles, then for any T_i, $T_j \in \mathcal{T}$ writing a common data item x and for any replicas $R^A, R^B \in \mathcal{R}$: $c_i^A \prec_t c_j^A$ in RH^A if and only if $c_i^B \prec_t c_j^B$ in RH^B. That is, two conflicting committed transactions commit in the same order in all local histories.

Proof. Assume two write transactions T_i and T_j updating the same data object, and two arbitrary replicas R^A and R^B. Since all local histories commit the same set of update transactions, we know that if c_i^B and c_j^B occur in RH^B so do c_i^A and c_j^A in RH^A and vice versa. Now assume $c_i^A \prec_t c_j^A$ in RH^A and $c_j^B \prec_t c_i^B$ in RH^B.

Let x be (one of) the objects that T_i and T_j both update. RH^A defines a total version order on x. Furthermore, since RH^A is a SI-history, based on Lemma 1, $c_i \prec_t c_j$ implies $x_i \ll x_j$. By the definition of write-dependency edges (as in Table 3.1), if $x_i \ll x_j$, then $SSG(RH^A)$, and thus $USG(RH)$, have a path $T_i \xrightarrow{ww^+} T_j$ consisting of only write-dependency edges . Similarly, $c_j^B \prec_t c_i^B$ in RH^B will lead to $T_j \xrightarrow{ww^+} T_i$ in $USG(RH)$. This results in $USG(RH)$ having a cycle consisting only of write-dependency edges. This contradicts the assumption that $USG(RH)$ avoids G-1c. □

Lemma 3 indicates that all replicas must commit conflicting update transactions in the same order.

Furthermore, Lemma 1 indicates that in an SI-history, the commit order of write transactions is consistent with the version order of the data items they write. Since each local history RH^k is a SI history, we can derive the following:

Proposition 2. *Let RH be a replicated history over $rmap(\mathcal{T}, \mathcal{R})$. At each $R^k \in \mathcal{R}$, let RH^k be a SI-history over $\mathcal{T}^k$. Let each update transaction $T_i \in \mathcal{T}$ commit at either all or none of the replicas.*

If $USG(RH)$ has no G-1c cycle, then for each $R^k, R^l \in \mathcal{R}$: $x_i \ll x_j$ in $RH^k \iff x_i \ll x_j$ in RH^l. That is, all local histories have the same version orders for all data items, and thus, the same write-dependency edges in their $SSG(RH^k)$.

Based on the discussion above, we can state sufficient and necessary conditions for a replicated history to be 1-copy-SI as follows.

Theorem 1. ***1-copy-SI Existence*** *Let RH be a replicated history over rmap($\mathcal{T}, \mathcal{R}$). RH is 1-copy-SI if and only if the following holds*

1. *For each $R^k \in \mathcal{R}$, RH^k is a SI-history.*
2. *For all update transactions $T_i \in \mathcal{T}$ and for all $R^k, R^l \in \mathcal{R}$: $c_i^k \iff c_i^l$.*
3. *$USG(RH)$ has no G-1c or G-SIb* cycles.*

Proof. To prove this, according to the definition of 1-copy-SI (Definition 7), it is sufficient to show that we are able to construct a SI-history H over $\mathcal{T}$ with the same write-, read-, and anti-dependencies as $USG(RH)$.

Part (1): To construct a history H.

To construct H over $\mathcal{T}$, we have to build the $\prec_t$-order of operations and the version order for all objects. For transactions that abort, we can take any execution order. We will make sure that nobody reads versions of aborted transactions to avoid G-1a. Then, we build a total order between start and commit operations of all committed transactions which reflect the $\prec_t$ order these operations have in H. This total order is derived from the dependencies in $USG(RH)$.

Step 1: Partially ordering starts and commits. In order to obtain this total order we construct a *Start-Commit-Order Serialization Graph*, $\mathcal{SCSG}(RH)$, in the following way.

1. The vertices of $\mathcal{SCSG}(RH)$ are the start and commit operations of all committed transactions (i.e., s_i and c_i for all T_i in $USG(RH)$).

2. For each T_i in $USG(RH)$, there is an edge $s_i \xrightarrow{\Re} c_i$ in $\mathcal{SCSG}(RH)$. This reflects the fact that the $\prec_t$-order requires the start of a transaction to be before its commit, i.e., $s_i \prec_t c_i$.

3. For each $T_i, T_j, i \neq j$ in $USG(RH)$ there is an edge $c_i \xrightarrow{\Re} s_j$ in $\mathcal{SCSG}(RH)$ iff $T_i \xrightarrow{e} T_j$ in $USG(RH)$ where $e \in \{ww, wr\}$. This reflects the fact that these dependencies imply $c_i \prec_t s_j$ in a SI-history[6].

4. For each $T_i, T_j, i \neq j$ in $USG(RH)$ there is an edge $s_i \xrightarrow{\Re} c_j$ in $\mathcal{SCSG}(RH)$ iff $T_i \xrightarrow{rw} T_j$. This reflects the fact that an anti-dependency implies $s_i \prec_t c_j$[7].

Now we show there is no cycle in $\mathcal{SCSG}(RH)$, and thus there is a partial order of start and commit operations. We do this by contradiction. Assume there is a cycle. It is important to note that all edges in $\mathcal{SCSG}(RH)$ are placed between start and commit operations (i.e., there are neither $s_i \xrightarrow{\Re} s_j$ nor $c_i \xrightarrow{\Re} c_j$ edges). Thus, without loss of generality, we can break the cycle into $m (m \geq 1)$ sections:

$c_{i_k} \xrightarrow{\Re} s_{j_k} \xrightarrow{\Re} c_{i_{(k+1)\%m}}$ (where $0 \leq k < m$)

In section k, the first edge $c_{i_k} \xrightarrow{\Re} s_{j_k}$ must be derived from a $T_{i_k} \xrightarrow{e} T_{j_k}$ ($e \in \{wr, ww\}$) in $USG(RH)$. The second edge $s_{j_k} \xrightarrow{\Re} c_{i_{(k+1)\%m}}$ must be derived either by (a) the $\prec_t$-order within a transaction, i.e., $j_k = i_{(k+1)\%m}$, or by (b) an anti-dependency between different transactions $T_{j_k} \xrightarrow{rw} T_{i_{(k+1)\%m}}$ ($j_k \neq i_{(k+1)\%m}$). We discuss all possibilities.

Assume that all edges of type $s_{j_k} \xrightarrow{\Re} c_{i_{(k+1)\%m}}$ are derived by (a) (i.e., $j_k = i_{(k+1)\%m}$), i.e., no edge was derived by an anti-dependency. Thus, the cycle in $\mathcal{SCSG}(RH)$ is due to a cycle in $USG(RH)$ that consists only of write- and read-dependency edges. However, $USG(RH)$ does not have G-1c cycles.

Therefore, there must be at least one section in the cycle such that $s_{j_k} \xrightarrow{\Re} c_{i_{(k+1)\%m}}$ is due to (b) (i.e., due to an anti-dependency). Note that $s_{j_k} \xrightarrow{\Re} c_{i_{(k+1)\%m}}$ must be prefixed with a $c_{i_k} \xrightarrow{\Re} s_{j_k}$ in the cycle. Thus, the cycle in $\mathcal{SCSG}(RH)$ must be due to a cycle in $USG(RH)$ with one or more

[6]Note that we assume that a transaction does not read its own writes and only writes an object once, therefore there is no $T_i \xrightarrow{ww,wr} T_i$ edge in $USG(RH)$.

[7]Note that for $T_i \xrightarrow{rw} T_i$ in $USG(RH)$ we have already $s_i \xrightarrow{\Re} c_j$ in $\mathcal{SCSG}(RH)$ due to step 1.

anti-dependencies where each anti-dependency is prefixed by a write- or read-dependency edge. This contradicts the fact that $USG(RH)$ has no G-SIb* cycles.

Step 2: Totally ordering starts and commits. $\mathcal{SCSG}(RH)$ so far defines a partial order between start and commit operations. We make this a total order (that is, connecting any start with any commit) in the following way: For any c_i, $s_j, i \neq j$ that are not connected in the graph (i.e., there is no path from c_i to s_j or from s_j to c_i), we set $s_j \xrightarrow{\Re} c_i$. This will not lead to any new cycles by construction.

Now we set the $\prec_t$-order between start and commit operations in $\mathcal{T}$ for our global history H according to $\mathcal{SCSG}(RH)$. For any aborted transaction T_i, we just order the s_i at the very beginning (as sources of $\mathcal{SCSG}(RH)$). We simply set a_i immediately after its s_i.

Step 3: Ordering write and read operations. Then we include the read and write operations of each committed transaction T_i into $\prec_t$ of H by setting them after s_i and before c_i according to the execution order within the transaction.

Step 4: Totally ordering versions of data items. We now have to determine the version order of all versions created by committed transactions. According to Proposition 2, all local histories RH^k at the different replicas have the same version orders for all data items. We will use these version orders for H.

Step 5: Determining the versions of read operations. Finally, we have to determine for each read operation on x, the version that is read. We simply do this in the following way. Let T_i have a read operation on x. Let T_j have a write operation on x, $c_j \prec_t s_i$ and there is no T_k, T_k also writes x, and $c_j \prec_t c_k \prec s_i$. Then we let T_i read x_j, i.e., we set $r_i(x_j)$ in H. If no such T_j exists, then we set $r_i(x_0)$ where x_0 is the last committed version of x before any transaction in $\mathcal{T}$ started.

Part (2): $SSG(H)$ has exactly the same write-, read- and anti-dependency edges as $USG(RH)$

Before we show that $SSG(H)$ and $USG(RH)$ have the same dependency edges, we show a useful property.

Lemma 4. *If $x_i \ll x_j$ in H, then $c_i \prec_t c_j$.*

Proof. Since H has the same version order as any local history RH^k, we have $x_i \ll x_j$ in RH^k and thus, there is a path $T_i, T_{k_1}, ...T_{k_n}, T_j$ in $USG(RH)$ consisting only of write-dependency edges. As

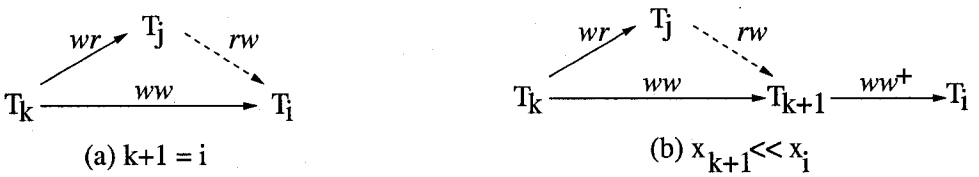


Figure 3.7: Proof of Theorem 1, Part (2), 1.(b), USG(RH) if $T_k \xrightarrow{wr} T_j$

a result of constructing $\mathcal{SCSG}(RH)$ and H, we have $c_i \prec_t s_{k_1} \prec_t c_{k_1} ... \prec_t s_{k_n} \prec_t c_{k_n} \prec_t s_j$, and thus $c_i \prec_t s_j \prec_t c_j$ in H. □

Write-dependency edges We first show that $SSG(H)$ and $USG(RH)$ have the same write-dependency edges. This is true, because in Step 3 above we build the version order of each data item in H to be the same as for any local history RH^k. A write-dependency edge is defined as a directed edge from one transaction installing a version of data item x to another transaction which installs the next version of x. Hence, $SSG(H)$ and $USG(RH)$ have the same write-dependency edges.

Read-dependency edges We now show that $SSG(H)$ and $USG(RH)$ have the same read-dependency edges.

1. We first show that whenever $T_i \xrightarrow{wr} T_j$ in $SSG(H)$ due to $r_j(x_i)$ in H, then $T_i \xrightarrow{wr} T_j$ in $USG(RH)$. Let T_j be a transaction local in R^l for RH. We show that in RH^l, T_j^l cannot read a later version than x_i nor an earlier version than x_i. Thus, it has to read x_i in RH^l and $T_i \xrightarrow{wr} T_j$ in $USG(RH)$.

   (a) Assume T_j^l reads x_k and $x_i \ll x_k$ in RH^l. Then $USG(RH)$ contains $T_k \xrightarrow{wr} T_j$. This results in $c_k \xrightarrow{\Re} s_j$, and thus, in $c_k \prec_t s_j$ in H. At the same time, $x_i \ll x_k$ results in $c_i \prec_t c_k$ in H according to Lemma 4. Combined, we have $c_i \prec_t c_k \prec_t s_j$ in H. In this case, however, when constructing H, according to step 5 of the construction of H, we would not have chosen T_j to read the version x_i because there is another transaction T_k that updated x and committed after T_j. Thus, there would be no $T_i \xrightarrow{wr} T_j$ in $SSG(H)$.

   (b) Assume T_j^l reads x_k and $x_k \ll x_i$ in RH^l. Then $USG(RH)$ contains $T_k \xrightarrow{wr} T_j$. Let x_{k+1} be the version that directly comes after x_k in the version order. At $SSG(RH^l)$,

there is a $T_j^l \xrightarrow{rw} T_{k+1}^l$ edge according to Proposition 1. Therefore, $USG(RH)$ has a $T_j \xrightarrow{rw} T_{k+1}$ edge (depicted in Figures 3.7.(a) and (b)). By construction, $\mathcal{SCSG}(RH)$ has a $s_j \xrightarrow{\Re} c_{k+1}$ edge and H has $s_j \prec_t c_{k+1}$.

If $k+1 = i$ (see Figure 3.7.(a)), this results in $s_j \prec_t c_i$ in H. If $x_{k+1} \ll x_i$ (see Figure 3.7.(b)), then we have $c_{k+1} \prec_t c_i$ in H according to Lemma 4, and thus again $s_j \prec_t c_i$. But with $s_j \prec_t c_i$, it is impossible for T_j to read x_i according to construction step 5. Therefore $USG(RH)$ cannot have $T_k \xrightarrow{wr} T_j$ and T_j^l cannot read x_k in RH^l.

(c) Finally assume that T_j^l reads a version x_0 that was committed before any of the transactions in $\mathcal{T}$ started. Then $USG(RH)$ will not have any read-dependency edge for T_j due to x. Let x_k be now the first visible version in the version order for x. Then there would be a $T_j^l \xrightarrow{rw} T_k^l$ in the $SSG(RH^l)$ of the local RH^l of T_j according to Proposition 1. Thus, a corresponding edge $T_j \xrightarrow{rw} T_k$ occurs in $USG(RH)$ and $s_j \xrightarrow{\Re} c_k$ in $\mathcal{SCSG}(RH)$, which results in $s_j \prec_t c_k$ in H.

Since x_k is the first visible version of x, we have either $k = i$ or $x_k \ll x_i$ in both RH^l and H. $k = i$ and $s_j \prec_t c_k$ imply $s_j \prec_t c_i$ in H. $x_k \ll x_i$ and $s_j \prec_t c_k$ imply $s_j \prec_t c_k \prec_t c_i$ in H. In both cases, $s_j \prec_t c_i$ in H. However, we already know that there is a $T_i \xrightarrow{wr} T_j$ in $SSG(H)$, which implies $c_i \prec_t s_j$. Therefore, T_j^l cannot read x_0.

As a result T_j^l must read x_i, leading to $T_i \xrightarrow{wr} T_j$ in $USG(RH)$.

2. Now we have to show that whenever $T_i \xrightarrow{wr} T_j$ in $USG(RH)$ due to $r_j^l(x_i)$ in the local history RH^l of T_j, then $T_i \xrightarrow{wr} T_j$ in $SSG(H)$. We have to prove that T_j can neither read a version before x_i (including x_0) nor a version after x_i in H. Hence, T_j must read x_i. The reasoning is very similar to above and omitted.

Anti-dependency edges Finally, since $SSG(H)$ and $USG(RH)$ have the same read- and write-dependency edges, based on Proposition 1, they must have the same anti-dependency edges.

Part (3): H is a SI history

G-1a and G-1b are avoided according to how aborted transactions are handled and how read operations are set. G-SIa is avoided by step 3 of constructing $\mathcal{SCSG}(RH)$, i.e., there is a $c_i \xrightarrow{\Re} s_j$

whenever there is a $T_i \xrightarrow{ww/wr} T_j$. Since $SSG(H)$ has the same write- and read-dependency edges as $USG(RH)$ and $USG(RH)$ has no G-1c cycle (containing only write- and read-dependency edges), H avoids G-1c.

Now we have to show that H avoids G-SIb. Note that we cannot only rely on $USG(RH)$ having no G-SIb* cycle, because $SSG(H)$ has more edges than $USG(RH)$, namely start-dependency edges, and thus, might still have a cycle containing an anti-dependency and some start-dependency edges. Thus, assume that $SSG(H)$ exhibits G-SIb (i.e., a cycle with one anti-dependency edge). Since whenever there is a write- or read-dependency edge from T_i to T_j, there is also a start-dependency edge, $SSG(H)$ must have a cycle $T_i \xrightarrow{S+} T_j \xrightarrow{rw} T_i$. Due to $T_i \xrightarrow{S+} T_j$, there must be $c_i \prec_t s_j$ in H. Due to $T_j \xrightarrow{rw} T_i$, T_j must read a data item x installed by another update transaction T_k, and T_i installs x right after T_k. Moreover, there must be $c_k \prec_t s_j \prec_t c_i$ in H, according to how the read operation $r_j(x_k)$ is ordered at the construction step 5 of H. $c_i \prec_t s_j$ due to $T_i \xrightarrow{S+} T_j$ and $s_j \prec_t c_i$ due to $T_j \xrightarrow{rw} T_i$ will derive $c_i \prec_t c_i$. This is impossible since we have totally ordered all start and commit operations in H at Step 2 above. Hence, H avoids G-SIb.

Thus, RH is 1-copy-SI. □

3.2.5 Observations

Lemma 3 indicates that all conflicting transactions must commit in the same order at all replicas. However, we have not discussed in what cases a transaction is allowed to commit. According to Snapshot-Write property of SI, if two transactions have write/write conflicts and are concurrent, one of them must be aborted. This rule also needs to hold in a replicated database. But when are two transactions concurrent in a distributed system? In a non-replicated system, two transactions T_i and T_j are concurrent if their lifetimes overlap (i.e., $s_i \prec_t c_j \wedge s_j \prec_t c_i$). We can define the concurrency of two transactions in a replicated database according to this rule.

Definition 8. *Let RH be a replicated history over $rmap(\mathcal{R}, \mathcal{T})$. Two transactions $T_i, T_j \in \mathcal{T}$ are concurrent in RH, iff $\exists R^k, R^l \in \mathcal{R}$: $s_i^k \prec_t c_j^k/a_j^k$ in RH^k and $s_j^l \prec_t c_i^l/a_i^l$ in RH^l.*

It means that T_i and T_j are concurrent if and only if T_i does not always start before T_j commits/aborts at all replicas (or vice versa). Note that R^k might be the same as R^l. It means that

if T_i and T_j are concurrent in one local history they are considered concurrent. But they are also considered concurrent if T_i executes completely before T_j in one history and completely after T_j in another history. Based on this definition, we can derive another rule for 1-copy-SI.

Theorem 2. *Let RH be a replicated history over $rmap(\mathcal{R}, \mathcal{T})$, and RH is 1-copy-SI. If two transactions $T_i, T_j \in \mathcal{T}$ have write/write conflicts and are concurrent in RH, at least one of them aborts.*

Proof. Assume both transactions commit.

Let's assume first that both transactions are concurrent at one local history RH^k, i.e., $s_i^k \prec_t c_j^k$ and $s_j^k \prec_t c_i^k$. That is, there is neither a start-dependency edge from T_i to T_j nor from T_j to T_i in $SSG(RH^k)$. Since they both write a common data item, $SSG(RH^k)$ must have a write-dependency edge from either T_i to T_j or vice versa. Thus, RH^k would violate G-SIa. But RH^k is a SI-history. Therefore T_i and T_j cannot be concurrent at any local history RH^k and commit.

Therefore, T_i and T_j must be concurrent because there are two local histories RH^k and RH^l, $k \neq l$, and $s_i^k \prec_t c_j^k$ and $s_j^l \prec_t c_i^l$. Furthermore, according to Lemma 3, both RH^k and RH^l must commit T_i and T_j in the same order. Assume without loss of generality, this order is $c_i \prec_t c_j$. This implies $s_i \prec_t c_j$. Therefore, at RH^l we have $s_i^l \prec_t c_j^l$ and $s_j^l \prec_t c_i^l$, meaning T_i and T_j are concurrent at RH^l which is impossible as shown above.

Therefore, T_i and T_j cannot be concurrent, have write/write conflicts and both commit. □

3.3 Snapshot isolation and integrity constraints

Database systems allow database designers to define a whole range of integrity constraints, such as primary keys and foreign keys. It is the task of database systems to enforce these constraints. In the next two sections, we discuss the relationship between snapshot isolation and integrity constraints. The current section focuses on a non-replicated system while the next extends our notions to a replicated environment.

3.3.1 Implementing integrity constraints

An integrity constraint puts constraints on the existence and values of data objects in the system. During the execution of a transaction these constraints might be violated. However, at the time of commit, all constraints must be obeyed. Many implementations, however, are pessimistic. That is, they never allow an update to occur that might violate the integrity constraints of the database.

The most simple constraint is the primary key constraint that disallows the existence of two records in a table with the same value in the primary key attribute. Before inserting a record, the system checks whether already a record with the same primary key value exists, and if yes, disallows the update and aborts the transaction. In the following, we will not further discuss this kind of constraint, because it is easy to detect and handle, and does not impose any problems in regard to SI.

The second most common constraint is the foreign key constraint, and we will use it as an example throughout the thesis[8]. Assume a relation *Dept(did, dname)* with the department identifier *did* as primary key, and a relation *Emp(eid, ename, did)* with the employee identifier *eid* as primary key and the attribute *did* as foreign key referring to the department the employee works in. The foreign key constraint requires that if there is an employee record with $did = x$ in the *Emp* table, then there is a department record in the *Dept* table with $did = x$.

In order to guarantee this property, a database system typically performs some implicit read operations upon receiving certain update requests. In above example, whenever a client wants

[8]More advanced constraints can be maintained via assertion or triggers. The principle is the same and we do not discuss them further in this thesis.

to insert an employee record or update the *did* field of an existing employee record, the system performs an implicit read operation on *Dept* to check whether a department record exists with the corresponding value in the *did* attribute. If it exists, the insert/update on *Emp* is allowed, otherwise it is forbidden and the transaction aborted. Similarly, if a client wants to delete a department record or set the *did* field of a department record to a different value, the system first looks at the *Emp* table and checks whether an employee record exists that has the same *did* value. If yes, the delete/update is rejected and the transaction aborted, otherwise the operation is allowed.[9]

The problem is that if these read operations run under snapshot isolation, integrity constraints could be violated.

Example 4. *Assume above tables* Dept(did, dname) *and* Emp(eid, ename, did). *Now assume a department record* ('d1','marketing') *already exists inserted by transaction* T_0 *and let's denote it with* x. *Now assume a transaction* T_1 *inserts an employee and transaction* T_2 *deletes the department.*

T_1*: insert into Emp values ('e1', 'Mike', 'd1');*

T_2*: delete from Dept where did='d1';*

We can denote the new employee as y. *Now assume a serial execution where* T_1 *runs before* T_2,

$$s_1, r_1(x_0), w_1(y_1), c_1, s_2, r_2(y_1), a2$$

That is, T_1 *reads the department tuple, determines that it exists, and performs the insert. After that* T_2 *first checks whether an employee exists, finds one, and thus, disallows the delete and aborts. In contrast, if* T_2 *runs before* T_1 *we have*

$$s_2, r_2(y_{init}), w_2(x_{dead}), c_2, s_1, r_1(x_{dead}), a_1$$

That is, T_2 *does not find any employee tuple and deletes the department. After that* T_1 *does not find a department tuple, disallows the insert of the employee and aborts.*

Now assume both transactions run concurrently and the read operations are performed on a snapshot:

$$H_{write-skew} : s_1, s_2, r_1(x_0), r_2(y_{init}), w_1(y_1), w_2(x_{dead}), c_1, c_2, [x_{dead}, y_1]$$

The read operation of T_1 *(i.e.,* $r_1(x_0)$*) finds a department with did='d1'. Hence,* T_1 *can continue*

[9]Note that SQL offers the definition ON DELETE CASCADE. In this case, if a transaction wants to delete a department tuple and there exist corresponding employee tuples, the transaction is not aborted but the employee tuples are also deleted. Similar semantics holds for ON UPDATE CASCADE. We do not consider this in this thesis.

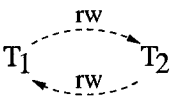


Figure 3.8: SSG($H_{write-skew}$) in Example 4

to insert the employee tuple. Similarly, the read operation of T_2 finds no employee associated with the department. Hence, T_2 can continue to delete the department. After both commit, the employee ('e1', 'Mike', 'd1') *refers to a non-existing department. Clearly this history does not respect foreign key constraints. However, $H_{write-skew}$ does not exhibit G-1 and G-SI. Therefore, $H_{write-skew}$ is a valid SI-history. Note that the only cycle in $SSG(H_{write-skew})$ (Figure 3.8) is a cycle with two adjacent anti-dependency edges.*

The problem is that reading from a snapshot is not the right thing to do for checking integrity constraints because it does not really help if the constraint holds at the beginning of the transaction. Instead, the constraint needs to hold at the time the transaction commits.

3.3.2 A new isolation level: SI+IC

Database systems that implement snapshot isolation guarantee that integrity constraints are not violated by distinguishing between standard read operations (that read from a snapshot) and read operations that are done to check constraint violations (that must get a more up-to-date state of the database system).

Thus, we model a new isolation level SI+IC that follows this model. It is stronger than the basic SI that we discussed in the last two sections, because it avoids integrity constraint violations. It is weaker than serializability because standard read operations continue to read from a snapshot. A SI+IC history should satisfy the following two requirements.

1. It should provide SI properties to operations not related to integrity constraints;
2. If a transaction commits, its updates do not violate the integrity of the database.

Our model assumes that all transactions perform, if necessary, read operations that check whether an update would violate the integrity of the database. That is, if a transaction performs a write operation that could potentially lead to the violation of the integrity of the database, the transaction

performs an *integrity read.* If the integrity read determines that integrity would be violated by the write operation, the transaction aborts. We denote an integrity read of T_i reading data item x installed by T_j as $ir_i(x_j)$. Of course, we have to rely on transactions to perform the proper actions upon integrity reads.

Definition 9. *We say a transaction T is IC-obeying, if it aborts when its integrity reads determine that one of T's write operations would lead to a violation of the integrity constraints of the database.*

In commercial systems the integrity read typically take place before the corresponding write operations or just at commit time (using deferred constraint checking). In theory, it could be any time during the execution of the transaction. The important issue is that the integrity constraint should hold at the time the transaction commits. That is, while the read takes place sometime *before* the commit, it should be still valid *at the time* of commit. That is, it is useless if a transaction T performs an integrity read on an object x, but the object x is overwritten before T commits. In $H_{write-skew}$ this is exactly what happens. T_2 finds no employee tuple but at its commit time a tuple exists. Thus, the integrity read of a transaction T should read the version of an object x, that reflects the latest committed version of x at the time T commits. We express this in the following way.

Definition 10. ***IC-Consistency.*** *Let H be a history over a set of transactions $\mathcal{T}$. Let $T_i \in \mathcal{T}$ perform an integrity read on x. We say an integrity read $ir_i(x_j)$ of a committed transaction T_i in H is* IC-consistent, *if*

1. $c_j \prec_t c_i$; *and*
2. *if* $x_j \ll x_k$, *then* $c_i \prec_t c_k$.

Property (1) guarantees that the read reflects a committed version at the time T_i commits. Property (2) guarantees that it is the latest committed version at T_i's commit time.

If all integrity reads of a transaction T are IC-consistent and T is IC-obeying, then it is guaranteed that the integrity constraints related to T's write operations hold when T commits.

Example 5. *Let's rewrite the $H_{write-skew}$ example above to*

$H_{not-IC} : s_1, s_2, ir_1(x_0), ir_2(y_{init}), w_1(y_1), w_2(x_{dead}), c_1, c_2, [x_{dead}, y_1]$.

$ir_2(y_{init})$ is not IC-consistent, because there is y_1, $y_{init} \ll y_1$ and $c_1 \prec_t c_2$.

In fact, our definition is somewhat stronger than what is needed. That means, integrity might be maintained even if the integrity read is not IC-consistent. Nevertheless, we will require all integrity reads to be IC-consistent because this allows to guarantee integrity constraints in a very simply way (and in fact that is what current database systems do).

Example 6. *As an example of that our definition is stronger than what is needed, assume that the database has a record* ('d1', 'marketing') *in the* Dept *table. Now assume two transactions*

T_1*: insert into Emp values ('e1', 'Mike', 'd1');*

T_2 *: update Dept set dname = 'marketing and sales' where did = 'd1';*

Let's refer to the department tuple as x, and to the employee tuple as y. Now assume the following history:

$H'_{not-IC} : s_1, s_2, ir_1(x_0), w_1(y_1), w_2(x_2), c_2, c_1\ [x_2, y_1]$

T_1 *reads the original department tuple and inserts the employee. Now T_2 changes the name of the department and commits (note that it does not need to perform any integrity read since it does not change the primary key of the department record) before T_1 commits. According to our definition $ir_1(x_0)$ is not IC-consistent, since a new committed version x_2 of x exists at the time T_1 commits. However, the integrity constraint itself still holds because x_2 only changed the name of the department and not its identifier.*

We now derive our new isolation level as follows.

Definition 11. ***Snapshot Isolation and Integrity Constraints (SI+IC).*** *A history H over a set of IC-obeying transactions $\mathcal{T}$ is a SI+IC-history if it fulfills the Snapshot-Read and Snapshot-Write properties (Definitions 1 and 2), and all integrity reads of committed transactions are IC-consistent.*

3.3.3 SI+IC in GID

We have seen in section 3.1 how we can check a set of phenomena (G-1, G-SI) to determine whether a history runs under SI. In this section, we show how we can extend the list of phenomena to check whether a history runs under SI+IC.

In order to capture integrity reads and their requirements in regard to the commit order, we introduce new dependencies and corresponding edges in the SSG of a history. We say T_j directly

Dependency Type	Description	SSG	Edge name
Directly IC-read-depends	T_i installs x_i and x_i is the same version of x in T_j's integrity read	$T_i \xrightarrow{wir} T_j$	IC-read-dependency edge
Directly IC-anti-depends	T_i performs an integrity read on x and T_j installs x's next version	$T_i \overset{irw}{-\!\!\rightarrow} T_j$	IC-anti-dependency edge
commit-depends	T_j commits after T_i commits (i.e., $c_i \prec_t c_j$)	$T_i \xrightarrow{C} T_j$	commit-dependency edge

Table 3.2: IC dependencies

IC-read-depends on T_i if it performs an integrity read that reads the version of an object created by T_i (i.e., $ir_j(x_i)$). We say T_j directly IC-anti-depends on T_i, if T_i performs an integrity read that reads a version of an object x and T_j creates x's next version in the version order. Finally, we say T_j commit-depends on T_i if T_i commits before T_j commits. These dependencies are summarized in Table 3.2. We have to extend the definition of SSG to include these new dependencies.

Definition 12. ***Start-ordered Serialization Graph (SSG).*** *The $SSG(H)$ of a history H over a set of IC-obeying transactions $\mathcal{T}$ is a directed graph where each node in $SSG(H)$ corresponds to a committed transaction in H, and there is a write-, read-, anti-, IC-read-, IC-anti-, commit-, or start-dependency edge from T_i to T_j if T_j directly write-, directly read-, directly anti-, directly IC-read, directly IC-anti, start-, or commit-depends on T_i, respectively.*

Given that the graph now contains more types of edges, the question is how much the phenomena G-1 and G-SI have to be adjusted to consider the new edges, and whether we have to add new phenomena. It turns out that we have to adjust very little. G-1 and G-SIa remain as they are. We only have to adjust G-SIb and add one new phenomenon:

- **G-SIb: Missed Effects.** A history H over a set of IC-obeying transactions $\mathcal{T}$ exhibits phenomenon G-SIb if $SSG(H)$ contains a directed cycle with exactly one anti-dependency edge that is prefixed by a write-, read-, or start-dependency edge. We refer to such cycle as a *G-SIb cycle.*

- **G-IC: IC Violation.** A history H over a set of IC-obeying transactions $\mathcal{T}$ exhibits phenomenon G-IC if $SSG(H)$ contains an IC-read or IC-anti-dependency edge from T_i to T_j without there also being a commit-dependency edge from T_i to T_j.

Note that we cannot capture the phenomenon that a transaction might not be IC-obeying. We have to trust that transactions are IC-obeying during their execution.

We now show that the avoidance of G1, G-SI and G-IC is sufficient and necessary for a history to be SI+IC.

Theorem 3. *A SI+IC history H over a set of IC-obeying transactions $\mathcal{T}$ avoids G-1, G-SI and G-IC.*

Proof. [2] contains the proofs that show that a history that fulfills Snapshot-Read and Snapshot-Write avoids G-1 and G-SIa. Since their definitions have not changed, we refer to the interested reader to [2]. Thus, we only need to prove that G-IC and the new definition of G-SIb are avoided.

Since all integrity reads in H are IC-consistent, $SSG(H)$ clearly avoids G-IC. An IC-read-dependency edge $T_i \xrightarrow{wir} T_j$ is derived from an $ir_i(x_j)$ in the history H. Since this read is IC-consistent, $c_i \prec_t c_j$ must hold in H, which implies a commit-dependency edge from T_i to T_j in $SSG(H)$. An IC-anti-dependency edge $T_i \overset{irw}{\dashrightarrow} T_j$ is derived from $ir_i(x_k)$, and x_j is the version following x_k in the version order, i.e., $x_k \ll x_j$ in H. Since the read is IC-consistent, $c_i \prec_t c_j$ must hold, which implies a commit-dependency edge from T_i to T_j.

Assume that G-SIb is not avoided. There will be a cycle in which the anti-dependency edge is prefixed by a write-, read-, or start-dependency edge. Since G-SIa and G-IC hold, there must also be a cycle that consists only of start- and commit-dependency edges and a single anti-dependency edge. That is, the cycle has the form

$(T_i \xrightarrow{S^*} T_j \xrightarrow{C^*} T_k)^* \xrightarrow{S^+} T_p \overset{rw}{\dashrightarrow} T_i$.

This implies $(c_i \prec_t s_j \prec_t c_j \prec_t c_k) \prec_t s_p \prec_t c_i$ in H which is impossible. Hence, G-SIb is avoided. □

Theorem 4. *If a history H over a set of IC-obeying transactions $\mathcal{T}$ avoids G-1, G-SI and G-IC, then it is a SI+IC-history.*

Proof. To prove this, we have to show that H fulfills the Snapshot-Read and Snapshot-Write properties and all its integrity reads are IC-consistent.

Assume there exists an integrity read that is not IC-consistent.

- The integrity read could violate property (1) of Definition 10, i.e., $ir_i(x_j)$ and $c_i \prec_t c_j$. However, then $SSG(H)$ would have a IC-read dependency edge $T_j \xrightarrow{wir} T_i$ and a commit-dependency edge $T_i \xrightarrow{c} T_j$ in the opposite direction, and thus would not avoid G-IC.

- The integrity read could violate property (2) of Definition 10, i.e., $ir_i(x_j)$ in H, and $x_j \ll x_k$ and $c_k \prec_t c_i$.

  - Let's first assume that x_k is the version following x_j. Thus, in $SSG(H)$ we have edges $T_i \xrightarrow{irw} T_k$, and $T_k \xrightarrow{c} T_i$ in the opposite direction. However, this would mean that there is an IC-anti-dependency edge from T_i to T_k without there also being a commit-dependency edge from T_i to T_k, thus G-IC is not avoided.
  - Now assume that $x_j \ll x_{j+1} \ll x_k$ holds. Then we have edges $T_i \xrightarrow{irw} T_{j+1}$, $T_{j+1} \xrightarrow{ww+} T_k$ and thus $T_{j+1} \xrightarrow{S+} T_k$ because of G-SIa, and $T_k \xrightarrow{c} T_i$. $T_{j+1} \xrightarrow{S+} T_k$ implies $c_{j+1} \prec_t c_k$ and we assume $c_k \prec_t c_i$. Thus, $SSG(H)$ contains an IC-anti-dependency edge $T_i \xrightarrow{irw} T_{j+1}$ but the commit-dependency edge goes in the other direction. Thus G-IC is not avoided.

For Snapshot-Read and Snapshot-Write, we use the proofs similar to those in [2]. Assume Snapshot-Write is not satisfied, because T_i and T_j both update data item x, they are concurrent and both commit. Without loss of generality, let's assume T_i commits before T_j. Then there is a write-dependency edge from T_i to T_j without a start-dependency edge in the same direction. It contradicts the avoidance of G-SIa. Assume Snapshot-Read is not satisfied.

- Snapshot-Read could be violated because T_i reads a data item (e.g., x) written by a concurrent transaction T_j (i.e., $r_i(x_j)$ and $s_i \prec_t c_j$). But this would mean $SSG(H)$ has a read-dependency edge from T_j to T_i without there being also a start-dependency edge, and H would not avoid G-SIa.

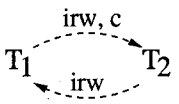


Figure 3.9: SSG($H'_{write-skew}$) in Example 7

- Snapshot-Read could also be violated because T_i reads data from a old snapshot instead of the latest snapshot, i.e., $r_i(x_j)$ and there is a $w_k(x_k)$, $c_k \prec_t s_i$ and $x_j \ll x_k$.
  - Assume x_k is the version directly following x_j. Due to Proposition 1, there is $T_i \overset{rw}{\longrightarrow} T_k$ in $SSG(H)$. According to our assumption $c_k \prec_t s_i$, we have $T_k \overset{S}{\rightarrow} T_i$ which leads to a cycle between T_i and T_k where the anti-dependency edge from T_i to T_k is prefixed by a start-dependency edge from T_k to T_i. Thus H would not avoid G-SIb.
  - Assume x_{j+1} is the version directly following x_j and $x_{j+1} \ll x_k$. Then, we have $T_i \overset{rw}{\longrightarrow} T_{j+1}$, $T_{j+1} \overset{ww+/S+}{\longrightarrow} T_k$ (due to G-SIa) and $T_k \overset{S}{\rightarrow} T_i$ (according to our assumption) again leading to a cycle with one anti-dependency which is prefixed by a start-dependency edge. Thus, again H would not avoid G-SIb.

□

Example 7. *Let's revisit Example 5.*

$H_{not-IC} : s_1, s_2, ir_1(x_0), ir_2(y_{init}), w_1(y_1), w_2(x_{dead}), c_1, c_2\ [x_{dead}, y_1]$,

$SSG(H_{not-IC})$ is shown in Figure 3.9. In the figure, the IC-anti-dependency edge from T_1 to T_2 is associated with a commit-dependency edge, but the other IC-anti-dependency edge is not. Hence, H_{not-IC} exhibits the G-IC phenomenon. As discussed in Example 5, it is not a SI+IC history because one of the integrity reads is not IC-consistent.

3.3.4 Observations

Above we have just shown that it is sufficient to show that if a history avoids G-1, G-SI and G-IC, then it is a SI+IC-history. Now we show that such a history avoids a further phenomenon:

- A history H over a set of IC-obeying transactions $\mathcal{T}$ exhibits phenomenon G-1c* if it contains a cycle that consists entirely of read-, write-, IC-read-, and IC-anti-dependency edges. We

refer to such a cycle as *G-1c* cycle.*

Lemma 5. *A SI-IC-history H avoids G-1c*.*

Proof. Assume it has such a cycle. Due to G-SIa and G-IC, there is also a cycle that consists only of commit- and start-dependency edges. This is impossible since each edge T_i to T_j in the cycle implies $c_i \prec_t c_j$, and thus transitively $c_i \prec_t c_i$. □

3.4 1-copy-SI+IC

In this section we extend our definition of 1-copy-SI to cover integrity constraints, denoting the new correctness criterion as 1-copy-SI+IC, and discuss sufficient conditions for a replicated history to be 1-copy-SI+IC.

A first issue is how to handle integrity reads in a replicated environment. Normal reads are executed at only one replica. If we do this for integrity reads, we easily end up with incorrect behavior. Let's revisit Example 5.

Example 8. *We let T_1 and T_2 of Example 5 execute in a replicated database with two replicas R^A and R^B. T_1 is submitted to R^A and T_2 is submitted to R^B. Recall there is a department record* ('d1', 'marketing') *and we refer to it as object x, version x_0. T_1 inserts a new employee (y) for x, and T_2 deletes x. Their integrity reads are only executed at their local replicas while their writesets are propagated to the other replica. We can get a RH with the following sub-histories.*

$RH^A : s_1^A, ir_1^A(x_0), w_1^A(y_1), c_1^A, s_2^A, w_2^A(x_2), c_2^A$

$RH^B : s_2^B, ir_2^B(y_{init}), w_2^B(x_2), c_2^B, s_1^B, w_1^B(y_1), c_1^A$

Since integrity reads exist only at the local replica, applying the write is not preceded by a check whether this write violates a constraint. Consequently, integrity constraints are violated in both sub-histories. In both histories, at the end of execution there is an employee tuple referring to a department that does not exist.

The problem is that the integrity read is something tightly related to the write operation. It checks something that has to hold in order for the write operation to be allowed to execute. One possibility to assure the proper behavior of the write is to perform the integrity read at all replicas. Therefore, we extend the ROWA mapper function of Definition 4 to include integrity reads at all replicas. We denote at IRS_i the set of all integrity read operations of transaction T_i.

Definition 13. ***Mapper function.*** *A ROWA mapper function, $rmap$, takes a set of IC-obeying transactions $\mathcal{T}$ and a set of replicas $\mathcal{R}$ as inputs, and transforms $\mathcal{T}$ into a set of transactions $\mathcal{T}' = rmap(\mathcal{T}, \mathcal{R})$. $rmap(\mathcal{T}, \mathcal{R})$ transforms each update transaction $T_i \in \mathcal{T}$ into a set of transactions $\{T_i^k | R^k \in \mathcal{R}\}$. In this set there is exactly one local transaction T_i^l where $WS_i^l = WS_i$,*

$IRS_i^l = IRS_i$ and $RS_i^l = RS_i$ (T_i is local at R^l). The rest are remote transactions T_i^r, where WS_i^r = WS_i, $IRS_i^r = IRS_i$ and $RS_i^i = \emptyset$ (T_i is remote at R^r). A read-only transaction T_i is transformed into a single local transaction T_i^l with $RS_i^l = RS_i$. We denote as $\mathcal{T}^k = \{T_i^k | T_i^k \in \mathcal{T}'\}$ the set of transactions executed at replica R^k.

From there we define 1-copy-SI+IC as below.

Definition 14. ***1-copy-SI+IC.*** *Let $RH = \bigcup RH^k, R^k \in \mathcal{R}$ be a replicated history over $rmap(\mathcal{T}, \mathcal{R})$. We say that RH is 1-copy-SI+IC if*

1. *For each $R^k \in \mathcal{R}$, RH^k is a SI+IC history;*
2. *For all update transactions $T_i \in \mathcal{T}$ and for all $R^k, R^l \in \mathcal{R}$: $c_i^k \iff c_i^l$;*
3. *There exists a global SI+IC history H over IC-obeying $\mathcal{T}$ such that*
   (a) *$SSG(H)$ and $USG(RH)$ have the same nodes*
   (b) *$SSG(H)$ has exactly the same read-, write-, and anti-dependency edges as $USG(RH)$.*

Note that IC-dependency edges are not considered in $USG(RH)$. Thus, local histories can have different integrity reads as long as all integrity reads have the same effect, i.e., either all local histories and the global history have integrity reads that allow the write operations to execute, and thus, the transaction commits, or all local histories and the global history have integrity reads that detect a violation, and thus, abort the transaction. Which version of a data item each of the histories reads is not relevant, as long as it has the same commit/abort effect as in the other histories. Let's have a look at an example.

Example 9. *Again assume a department record* ('d1', 'marketing') *exists inserted by transaction T_0 and let's denote it with x_0. Now assume a transaction T_1 inserts an employee y referring to x and T_2 renames the department.*

T_1: insert into Emp values ('e1', 'Mike', 'd1');

T_2: update Dept set dname='accounting' where did='d1';

Note that T_2 does not have any integrity read at all. T_1 is submitted to R^A and T_2 is submitted to R^B. We can get the following replicated history.

(a) SSG(RH^A) (b) SSG(RH^B)

Figure 3.10: SSGs of Example 9

$RH^A : s_1^A, ir_1^A(x_0), w_1^A(y_1), c_1^A, s_2^A, w_2^A(x_2), c_2^A$
$RH^B : s_2^B, w_2^B(x_2), c_2^B, s_1^B, ir_1^B(x_2), w_1^B(y_1), c_1^B$

T_1 performs an integrity read on x_0 at R^A, and on x_2 at R^B. In both cases, the subsequent write (insert of employee) can succeed. $SSG(RH^A)$ and $SSG(RH^B)$ are shown in Figure 3.10.(a) and (b), respectively. At the commit time of any of the transactions, no integrity constraint is violated. The $USG(RH)$ only contains T_1 and T_2 but no edges. A global history could be equivalent to either RH^A or RH^B. Although the two transactions are indirectly ordered in opposite order in the two histories due to the integrity read which reads different versions at the different replicas, from an abstract point of view, this does not matter, as long as both integrity reads lead to the same commit/abort decision for the transaction.

While this definition is very flexible and does not restrict the execution in the histories unnecessarily, it makes it very hard to come up with conditions that are both sufficient and necessary for a replicated history to be 1-copy-SI+IC. Let's have a look at another example.

Example 10. *Assume a database with three tables:* Dept(did, dname), Emp(eid, ename, did), Stats(year, month, numberempl). *In the first setting assume a department record* ('d1', 'marketing') *denoted as x already exists and created by transaction T_0. There are the following three transactions:*

T_1: insert into Emp values ('e1', 'Mike', 'd1');

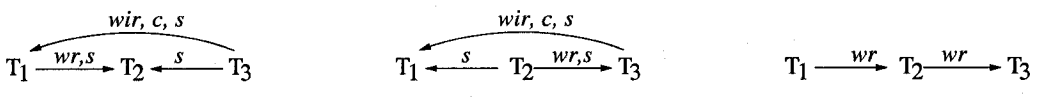
T_2: select count(*) from Emp (into program variable z);

insert into Stats values (2007, 07, z);

*T_3: select * from Stats;*

update Dept set dname = 'accounting' where did='d1';

We denote the new employee as y and the new Stats *record as z. Now assume that T_1 and T_2 are local at R^A and T_3 is local at R^B, and R^A executes the transactions serially in order T_3, T_1 and*



(a) SSG(RH^A) (b) SSG(RH^B) (c) USG(RH)

Figure 3.11: SSG and USG of Example 10

T_2, while R^B executes them serially in order T_2, T_3 and T_1. More formally the replicated history is:

$RH^A : s_3^A, w_3^A(x_3), c_3^A, s_1^A, ir_1^A(x_3), w_1^A(y_1), c_1^A, s_2^A, r_2^A(y_1), w_2^A(z_2), c_2^A$

$RH^B : s_2^B, w_2^B(z_2), c_2^B, s_3^B, r_3^B(z_2), w_3^B(x_3), c_3^B, s_1^B, ir_1^B(x_3), w_1^B(y_1), c_1^B$

The $SSGs$ of RH^A and RH^B are shown in Figures 3.11 (a) and (b) respectively. $USG(RH)$, shown in Figure 3.11 (c), does not have any cycles. Can we find a global history H that has the same dependency edges? Yes, we can. Since there are read-dependency edges from T_1 to T_2, and from T_2 to T_3, the only possible global history is a serial execution of T_1, T_2 and T_3, i.e.,

$H : s_1, ir_1(x_0), w_1(y_1), c_1, s_2, r_2(y_1), w_2(z_2), c_2, s_3, r_3(z_2), w_3(x_3), c_3.$

In this history, the same standard read operations are performed as in RH^A and RH^B. However, the integrity read reads a different version (x_3 in RH^A and RH^B while it is x_0 in H). Having a different read is fine, because the only condition for the integrity read is the existence of a department record x, independently of the name of the department. That is, although the global history reads a different version of x than the local histories, the transaction remains IC-obeying.

However, if we change the example slightly, this is no more the case. Assume no department with did='d1' exists at the beginning (i.e., x_{init}) in the database, and T_3, instead of renaming the department, actually inserts the department, i.e., the second operation of T_3 is "insert into Dept values ('d1', 'marketing');". Now assume that the execution at R^A and R^B is in the same order as above. In this case, the $SSGs$ and $USG(RH)$ remain the same as in Figures 3.11.(a)-(c), requiring the global history H to execute serially T_1 before T_2 before T_3. However, in this case, T_1 would not be IC-obeying since its integrity read is $ir_1(x_{init})$ indicating a violation of integrity constraints, nevertheless the transaction commits. Thus, in order to be IC-obeying, T_1 would need to abort, and thus $SSG(H)$ would not have the same dependency edges as $USG(RH)$.

The issue with the above examples was that a global history with the same dependencies as

$USG(RH)$ might require a transaction T to perform the integrity read on a data version that was different to any data version read in any of the local histories. Sometimes this integrity read can be valid, but other times it might lead to an abort. But this is very application dependent, and thus, cannot be captured by the formalism.

Nevertheless, we can give a sufficient condition for a replicated history to be 1-copy-SI+IC. The example above will not fulfill this condition. The idea is the following. We need to construct a SI+IC-history H that has the same read, write-, and anti-dependency edges as $USG(RH)$, and for every committed transaction T_i, there exists at least one replica R^k, such that the integrity reads of T_i in H read exactly the same data versions than in RH^k. That is, if $ir_i^k(x_j)$ in RH^k, then $ir_i(x_j)$ in H. Since RH^k is a SI+IC-history over IC-obeying transactions, we know that this integrity read $ir_i^k(x_j)$ indicated that the write that depends on this integrity read does not violate the integrity of the database. If we can find such a SI+IC history, we know that all its transactions are IC-obeying, and thus, RH is 1-copy-SI+IC. Note that we allow different transactions to have integrity reads from different replicas, e.g., T_i can have the same integrity reads as in RH^k, while T_j has the same integrity reads as in RH^l. But we require all integrity reads of an individual transaction T_i to be taken from one local history because they might be related to each other (e.g., the sum of x and y may not be below 100).

Further note that this condition can only be sufficient but is not necessary. In the first part of Example 10, RH is 1-copy-SI+IC but the only global history with the same read-, write-, and anti-dependency edges as $USG(RH)$ performs an integrity read on x that is different from the integrity read of any local history. The problem is that without application knowledge it is not clear whether the transaction remains IC-obeying. From here, we define an extended USG of a replicated history.

Definition 15. ***Union Serialization Graph with Integrity Dependencies (USG-IC).*** *Let $RH = \bigcup RH^k$ be a replicated history over $rmap(\mathcal{T}, \mathcal{R})$. We denote as USG-IC(RH) the following graph.*

1. *For each $R^k \in \mathcal{R}$, if $SSG(RH^k)$ has node $T_i^k \in \mathcal{T}^k$, then USG-IC(RH) has a node T_i.*
2. *For each $R^k \in \mathcal{R}$ and each write-, read-, or anti-dependency edge from T_i^k to T_j^k in $SSG(RH^k)$, USG-IC(RH) has a corresponding write-, read-, or anti-dependency from T_i to T_j.*

3. *For each $T_i \in \mathcal{T}$, there exists $R^k \in \mathcal{R}$, each IC-anti-dependency edge from T_i^k to T_j^k and each IC-read-dependency edge from T_j^k to T_i^k in $SSG(RH^k)$ has a corresponding IC-anti-dependency edge from T_i to T_j or IC-read-dependency edge from T_j to T_i in USG-IC(RH).*

4. *There are no further edges or nodes in USG-IC(RH).*

Note there is no unique *USG-IC(RH)* since there can be many combinations of choosing a local history RH^k for a transaction T_i. That is, if there are n replicas and t transactions there could be as many as n^t different *USG-IC(RH)*.

Theorem 5. ***1-copy-SI+IC Existence*** *Let RH be a replicated history over rmap($\mathcal{T}, \mathcal{R}$) with the following properties:*

- *For each $R^k \in \mathcal{R}$, RH^k is a SI+IC-history;*
- *For all update transactions $T_i \in \mathcal{T}$ and for all $R^k, R^l \in \mathcal{R}$, $c_i^k \Longleftrightarrow c_i^l$;*
- *There exists a $USG\text{-}IC(RH)$ that has no G-1c* or G-SIb* cycles.*

Then RH is 1-copy-SI+IC.

Proof. The proof is similar to the one for Theorem 1. We have to construct a SI+IC-history H with the same nodes and the same write-, read-, and anti-dependency edges as $USG(RH)$.

Part (1): To construct a history H.

We first build the same *Start-Commit-Order Serialization Graph*, $\mathcal{SCSG}(RH)$, from $USG\text{-}IC(RH)$ as described in the proof of Theorem 1 which provides a partial order between pairs of start- and commit operations. However, we add additional edges:

5. For each T_i, T_j ($i \neq j$) in *USG-IC(RH)*, there is an edge $c_i \xrightarrow{\Re} c_j$ in $\mathcal{SCSG}(RH)$ iff $T_i \xrightarrow{e} T_j$ in *USG-IC(RH)* where $e \in \{wir, irw\}$. This reflects the need that integrity reads need to be IC-consistent.

Now we show there is no cycle in $\mathcal{SCSG}(RH)$. Assume that there is a cycle. The cycle in $\mathcal{SCSG}(\mathcal{RH})$ consists either (a) entirely of commit operations or (b) of start and commit operations.

For case (a), there will be a corresponding cycle in *USG-IC(RH)* that consists entirely of IC-read- and IC-anti-dependency edges. This contradicts the fact that *USG-IC(RH)* has no G-1c* cycle.

For case (b), since there is no $s_i \xrightarrow{\Re} s_j$ in $\mathcal{SCSG}(RH)$, we can break the cycle into sections with either (i) the pattern of $c_i \xrightarrow{\Re^+} c_j$, or (ii) the pattern of $c_i \xrightarrow{\Re} s_j \xrightarrow{\Re} c_k$.

Pattern (i) $c_i \xrightarrow{\Re^+} c_j$ is due to a path of IC-read and IC-anti-dependency edges from T_i to T_j in *USG-IC(RH)*. In pattern (ii), $c_i \xrightarrow{\Re} s_j$ must be due to $T_i \xrightarrow{wr,ww} T_j$. $s_j \xrightarrow{\Re} c_k$ might be because s_j and c_k are in the same transaction (i.e., j=k) or because of $T_j \xrightarrow{rw} T_k$. If all dependencies $s_j \xrightarrow{\Re} c_k$ are due to j=k, we know that there is no anti-dependency edge in the cycle. The cycle must consist entirely of read-, write-, IC-read-, and IC-anti-dependency edges. It contradicts the fact that *USG-IC(RH)* has no G-1c* cycles. If some dependencies $s_j \xrightarrow{\Re} c_k$ are due to $T_j \xrightarrow{rw} T_k$, we know that each must be preceded by a $c_i \xrightarrow{\Re} s_j$ that was due to a $T_i \xrightarrow{wr,ww} T_j$. Thus, there must be a cycle in *USG-IC(RH)* such that all of its anti-dependency edges are prefixed with a read-, or write dependency edge. This contradicts the fact that *USG-IC(RH)* has no G-SIb* cycles.

Thus, our extended $\mathcal{SCSG}(RH)$ does not contain any cycles. We can construct H by using the four steps in the proof part (1) of Theorem 1. Additionally, we add two steps, i.e., Step 1.5 between step 1 and 2, and Step 6 after Step 5.

Step 1.5: Totally ordering commits. $\mathcal{SCSG}(RH)$ defines so far a partial order between commit operations. We extend this to a total order in the following way. For any c_i and c_j that are not connected in $\mathcal{SCSG}(RH)$, we set either $c_i \xrightarrow{\Re} c_j$ or $c_j \xrightarrow{\Re} c_i$.

Step 6: Determining the versions of integrity read operations. We need to determine for each integrity read operation on x, the version that is read. Let T_i have an integrity read on x. Let T_j have a write operation on x, $c_j \prec_t c_i$ and there is no T_k, T_k also writes x, and $c_j \prec_t c_k \prec_t c_i$. Then we let T_i read x_j, i.e., we set $ir_i(x_j)$ in H. If no such T_j exists, then we set $ir_i(x_0)$ where x_0 is the last committed version of x before any transaction in $\mathcal{T}$ starts.

Part (2): $SSG(H)$ has exactly the same read-, write-, and anti-dependency edges as USG(RH)

This part of the proof is the same as Theorem 1 proof part (2).

Part (3): H is a SI+IC history

The part of the proof that shows that H is a SI-history is similar to the proof of Theorem 1, part (3), and thus, is omitted here.

By construction step 6, it is clear that each integrity read is IC-consistent because the version to be read has been determined according to the definition of IC-consistency.

What remains to be shown is that in this artificially generated history transactions are actually IC-obeying. We do this by showing that for each transaction T_i and its integrity reads, there exists a RH^k such that if $ir_i^k(x_j)$ occurs in RH^k, then $ir_i(x_j)$ occurs in H. RH^k is a SI+IC history where the transaction really executed, and thus, we know that it is IC-obeying. Thus, if we take all the integrity reads of T_i from this history, we can be sure that T_i's write operations do not cause an integrity constraint violation.

We show this in the following way. Let R^k be the replica such that *USG-IC(RH)* took its IC-read- and IC-anti-dependency edges for T_i from $SSG(RH^k)$. Let $ir_i^k(x_j)$ occur in RH^k. We show that $ir_i(x_j)$ occurs in H.

If $ir_i^k(x_j)$ occurs in RH^k, then $SSG(RH^k)$, and thus *USG-IC(RH)*, have a IC-read-dependency edge from $T_j \xrightarrow{wir} T_i$, and if there exists a version x_{j+1} following x_j, then there is also an IC-anti-dependency edge from $T_i \xrightarrow{irw} T_{j+1}$. This leads to $c_j \xrightarrow{\Re} c_i \xrightarrow{\Re} c_{j+1}$ in $\mathcal{SCSG}(RH)$, and thus $c_j \prec_t c_i \prec_t c_{j+1}$ in H.

Assume now that in H, T_i reads an earlier version, i.e., either x_0 or a x_k such that $x_k \ll x_j$. In this case, according to the construction of integrity read, we have $c_i \prec_t c_j$ which is a contradiction to above requirement of $c_j \prec_t c_i$. Assume now that in H, T_i reads a later version, i.e., any x_k such that $x_j \ll x_k$ which means $c_k \prec_t c_i$. If x_k is the same as x_{j+1} we have a contradiction to above requirement $c_i \prec_t c_{j+1}$. If $x_{j+1} \ll x_k$, then $c_{j+1} \prec_t c_k \prec_t c_i$ again leading to a contradiction to above $c_i \prec_t c_{j+1}$. Therefore, T_i must read $ir_i(x_j)$ in H. □

Chapter 4

Replica control basics

As mentioned at the end of Chapter 2, our replication tool has to fulfill a wide range of properties. It has to provide 1-copy-SI+IC. It has to work at the middleware level in order to exist as an independent component and work with a heterogeneous environment. It should not pose any specific requirements to the applications but work with any kind of legacy application. That is, it should not require to mark transactions as read-only or update as in primary copy approaches, or require to know all operations of a transaction in advance. It should provide concurrency control at the record level and not the table level or other coarser concurrency levels. It should be fault-tolerant. It should work well even if message latencies are high, i.e., in wide-area networks (WANs).

In order to keep the description simple and understandable, the replication solution proposed in this thesis is developed incrementally. This chapter proposes protocols that guarantee 1-copy-SI/1-copy-SI+IC. However, it is not concerned with message overhead and fault-tolerance. The next chapter then extends these protocols in order to reduce the message overhead, and make them fault-tolerant.

All the protocols in this thesis assume that the underlying database systems provide SI+IC as discussed in Section 3.3. The protocols will not work for database systems that provide standard serializability and use strict 2PL.

The protocols of this chapter all assume a centralized middleware architecture as depicted in Figure 2.1.(a). There is one middleware instance and a set of databae replicas $\mathcal{R}$.

In the following we present a *Simple Replication Protocol*, SRP, which guarantees 1-copy-SI and provides a standard database interface to the application. However, SRP has deadlock problems when used on top of some database systems. Besides, SRP only guarantees 1-copy-SI and does not work for databases with integrity constraints. Hence, we propose SRP-IC. SRP-IC is based on SRP and guarantees 1-copy-SI+IC. It also handles deadlocks.

4.1 Simple Replication Protocol (SRP)

4.1.1 Basic idea

Our first protocol, SRP, provides 1-copy-SI, i.e., integrity constraints are not considered.

The protocol skeleton

We explain the basic idea of our protocol in terms of the lifetime of a transaction. A client submits the operations of a transaction T one by one to the middleware. When the middleware receives the start operation of T, it assigns a database replica $R^l \in \mathcal{R}$ to T and starts T at R^l. R^l is called the local database replica of T. The middleware then simply forwards all read and write operations of T to R^l. R^l executes the operations locally and returns the results to the middleware that forwards them to the client that submits T. At the end of the transaction, if the client requests an *abort*, the middleware simply asks R^l to abort T, and then returns the confirmation result to the client. If the client requests a *commit*, the middleware extracts the writeset of T from R^l. The writeset contains the physical changes made by T in R^l and the primary keys of all modified tuples[1].

The middleware then performs a validation test for T based on the writeset. A successful validation test will lead to the commit of T, and an unsuccessful test to its abort. The validation assures the execution is 1-copy-SI. Validation of transaction T will succeed if no transaction T' that validated before T and was concurrent to T had a write/write conflict. If such a transaction exists the validation of T fails. That is, in our protocol, if any two concurrent transactions have write/write

[1]Writeset extraction is a standard mechanism in many commerical replication solutions (e.g., [76]) implemented via triggers or log-sniffing. Although commercial systems usually export writesets only after commit, the functionality per se exists. We provide a pre-commit extraction similar to the ones developed in other research prototypes [61, 90, 94].

conflicts, the first to request commit will succeed, the other will abort. That is, we follow the first-committer-wins strategy. We defer the details of the validation to later.

If the validation fails, the middleware tells R^l to abort the transaction. Otherwise, it applies the writeset of T at all replicas (except R^l) and makes sure that all commit transactions in validation order.

Let's make some observations here. Firstly, we perform validation only after the entire transaction has executed. At this time we know exactly the records the transaction has updated. Thus, we avoid the limitations of previous middleware-based update everywhere approaches that perform the synchronization at transaction start time and thus, require to know all operations in advance – which is difficult if execution is non-deterministic and often only allows conflict detection at the table level. Furthermore, SRP commits all update transactions at all replicas in the same order. Thus, all conflicting transactions commit at all replicas in the same order – which is required for 1-copy-SI according to Lemma 3.

Validation

Let's now come back to validation. At the time of validation of transaction T_i, if it has a write/write conflict with a concurrent transaction T_j that validated before T_i, then T_i must abort. We can easily determine whether two transactions conflict by checking whether the sets of primary keys contained in their writesets overlap.

In order to determine whether a previously validated transaction T_j is concurrent to T_i we use timestamps. The middleware keeps for each database replica R^k a logical clock. Every time a transaction commits at R^k its logical clock is incremented by one. The middleware also keeps a validation clock. When a transaction T is successfullly validated, the value of the validation clock is assigned to T as tid-timestamp, and then the validation clock is incremented.

Since all database replicas commit transactions in validation order, the timestamp of a database replica R^k is the same as the tid of the latest committed transaction at R^k. For example, immediately after a transaction with $tid = 5$ commits at R^k, R^k's clock is 5. Transactions have additionally a *start* timestamp. If R^k is T's local replica, then $T.start$ is assigned the value of R^k's logical clock at the time T starts. With this, we know that a previously validated transaction T_j is *not* concurrent to T_i, if $T_j.tid \leq T_i.start$ because then T_i started at its local replica only after T_j committed. Thus,

Initialization:
$next\_tid := 1$, $ws\_list := \{\}$
$\forall\ R^k$: $tocommit\_queue_k := \{\}$
$\forall\ R^k$: $lastcommitted\_tid\_k := 0$
$wsmutex$,
$\forall\ R^k$: $dbmutex\_k$

1. Upon receiving an operation Op_i of T_i

(a) if Op_i is start (i.e.,s_i), then
  i. choose R^k at which T_i will be local
  ii. obtain $dbmutex\_k$
  iii. $T_i.start := lastcommitted\_tid\_k$
  iv. begin T_i^k at R^k
  v. release $dbmutex\_k$
  vi. return to client

(b) else if Op_i is read or write, then
  i. execute at local R^k and return to client

(c) else if Op_i is abort, then
  i. abort T_i^k at R^k and retun to client

(d) else (commit)
  i. $T_i.WS := getwriteset(T_i^k)$ from local R^k
  ii. if $T_i.WS = \emptyset$, then
    - commit and return
  iii. obtain $wsmutex$
  iv. if $\nexists T_j \in ws\_list$ such that $T_i.start < T_j.tid \wedge T_i.WS \cap T_j.WS \neq \emptyset$:
    - $T_i.tid := next\_tid++$
    - append T_i to ws_list
    - $\forall R^k$: append T_i to $tocommit\_queue\_k$
    - release $wsmutex$
  v. else
    - release $wsmutex$
    - abort T_i^k at R^k

2. Upon T_i is first in $tocommit\_queue\_k$

(a) if T_i is remote at R^k, then
  - begin T_i^k at R^k
  - apply $T_i.WS$ to R^k

(b) obtain $dbmutex\_k$

(c) commit at R^k

(d) $lastcommitted\_tid\_k++$

(e) release $dbmutex\_k$

(f) if local, return to client

(g) remove T_i from $tocommit\_queue\_k$

Figure 4.1: SRP: a Simple Replication Protocol

T_j is concurrent to T_i if $T_j.tid > T_i.start$.

4.1.2 Protocol details

The details of SRP are shown in Figure 4.1. We assume n database replicas R^k, $1 \leq k \leq n$. We assume all replicas provide SI using the first-committer-wins rule. That is, validation of write/write conflicts is only done at the commit time of a transaction. Such validation will fail if any concurrent transactions have been validated successfully and have write/write conflicts with the current transaction. Otherwise it will succeed. All start, read, write, commit and abort operations are

submitted to the middleware. $next\_tid$ represents the validation clock. The middleware maintains a list of already validated transactions (*ws_list*). Although all successfully validated transactions will be committed at the different database replicas in validation order the replicas might run at different speed. Hence, the middleware keeps for each replica R^k a queue *tocommit_queue_k* which contains the writesets to be executed and committed at R^k, and a logical clock *lastcommitted_tid_k* indicating the tid of the last committed transaction at R^k. If actions of different transactions need to be synchronized, appropriate mutexes are acquired.

Upon the start of a new transaction T_i (step 1a) one database replica is chosen to be the local replica. Before the start of T_i at the database replica, we get a mutex that avoids that the start operation is concurrent with any commit operations at the replica. Then we set $T_i.start$ to the tid-value of the last transaction that committed at R^k. As we discussed earlier, this allows us to determine concurrent transactions. We denote as T_i^k the incarnation of T_i at replica R^k. Read and write operations are then simply forwarded to the database replica. Since we assume the database replica to provide SI, T_i^k reads from a snapshot and writes new object versions (1b). If the operation is abort, the middleware simply forwards it to R^k and let R^k abort the transaction locally (1c). A confirmation message is returned to the corresponding client.

If the client requests a commit, actions are more complex. The middleware first retrieves the writeset from the local replica (1d.i). If it is empty, T_i is a read-only transaction and can simply be committed locally (1d.ii). Otherwise, the middleware starts a validation phase (1d.iii-v). Only one transaction can be in validation phase. Therefore we set a mutex, i.e., $wsmutex$. T_i's writeset is compared against all writesets of concurrent transactions that validated before (maintained in ws_list). As mentioned before, T_i is concurrent to a previously validated transaction T_j if $T_i.start < T_j.tid$, and it conflicts with T_j if their writesets overlap. If there is no concurrent conflicting and validated transaction, transaction T_i receives its tid value, and its writeset is added to all queues (*tocommit_queue_k* and *ws_list*). Otherwise, T_i aborts (1d.v). Writesets will be applied in the same order but at different speeds at individual replicas (step 2). At the local replica, of course, the writeset does not need to be applied. Still, the commit order in regard to other transactions must be maintained. Hence, the local transaction only commits when all the writesets stored in the queue at the time of validation have been applied. Whenever a transaction commits at a replica R^k,

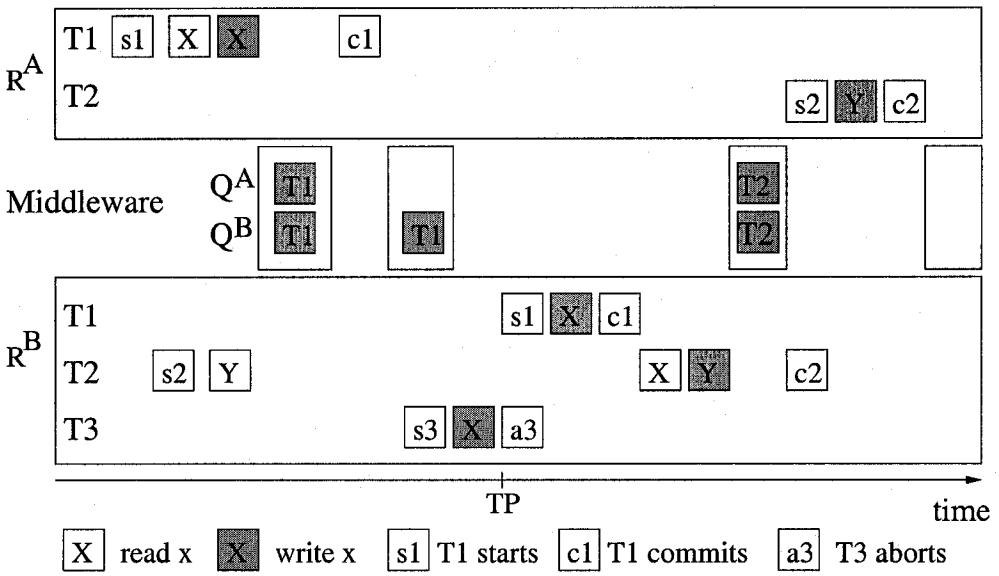


Figure 4.2: SRP Sample execution

$lastcommitted\_tid\_k$ is incremented. As mentioned before, committing a transaction at replica R^k (steps 2b-e) and starting a transaction (steps 1a.ii-v) are mutually exclusive. In summary, validation phase starts when the writeset is retrieved, validation is an atomic process but runs concurrently to committing and applying the writesets. However, applying writesets by itself occurs again in a serial fashion. Note that in order for clients to read their own writes, a transaction should only be assigned to a replica if all previous transactions of the same client are already committed at this replica.

4.1.3 Example

Example 11. *Figure 4.2 shows an example. The set of transactions is* $\mathcal{T} = \{T_1 = (s_1, r_1(x), w_1(x), c_1), T_2 = (s_2, r_2(y), r_2(x), w_2(y), c_2), T_3 = (s_3, w_3(x), c_3)\}$. T_1 *is local at replica* R^A, *and* T_2 *and* T_3 *are local at replica* R^B. *In the figure, grey boxes reflect writes, and white boxes represent reads, start, abort, or commit. The middleware keeps* tocommit_queue *for each replica (*Q^A *and* Q^B*). The figure shows the temporal evolution of the queues and transaction execution from left to right.*

T_1 *starts at* R^A *and reads and updates* x. *At* R^B, T_2 *starts and reads* y. *Upon* T_1*'s commit request, the middleware retrieves the writeset, validation succeeds, and* T_1 *receives* $T_1.tid = 1$. T_1 *is appended to* Q^A *and* Q^B. *Since* T_1 *is the first in* Q^A, T_1 *commits at* R^A *and is removed from* Q^A

*(*lastcommitted_tid_A = 1*). T_3 now starts at R^B. Although T_3 begins after T_1 commits in R^A, it is concurrent to T_1 in R^B since T_1's updates are not yet applied in R^B. Hence, $T_3.start = 0$. When T_3 now submits commit at timepoint TP, T_3's validation at the middleware fails since $T_3.start = 0 < T_1.tid = 1$ and the writesets overlap. Hence, T_3 is aborted at R^B. At the same time, R^B applies T_1's writeset and commits T_1. T_1 is removed from Q^B and* lastcommitted_tid_B *set to 1. Although T_2's read is after T_1's write it does not read the value written by T_1 since the transactions are concurrent in R^B. After T_2's execution its validation succeeds ($T_2.tid = 2$) since it has no write/write conflict with T_1. T_2 is appended to Q^A and Q^B and later committed at both replicas.*

4.1.4 Correctness

Theorem 6. *SRP provides 1-copy-SI if the underlying database replicas provide SI using the first-committer-wins rule.*

Proof. Based on Theorem 1, we need to prove that for any replicated history RH possible under SRP, (i) the local history RH^k at all replicas are SI-histories, (ii) a write transaction commits at either none or all replicas, (iii) the corresponding $USG(RH)$ has no G-1c and G-SIb* cycles.

Property (i) is fulfilled since the underlying database replicas provide SI by assumption.

For property (ii), we need to show that, apart of read-only transactions, all replicas commit the same set of transactions. If validation of a transaction T_i succeeds at the middleware it is appended to $tocommit\_queue\_k$ of each replica R^k. Transactions in $tocommit\_queue\_k$ are handled one after the other. Let T_i be the first in the queue. If T_i is a remote transaction, no other transaction commits between T_i's start and T_i's commit at R^k. Since we assume that the underlying database uses the first-committer-wins rule, there is no concurrent transaction that validates before T_i. Thus, T_i's validation within the database R^k will succeed and T_i will commit. If T_i is local at R^k, then T_i has already started at R^k. If T_i conflicted with any transaction (local or remote) that committed at R^k since T_i's start, R^k would abort T_i when the commit request is submitted since R^k provides SI. But at validation, the middleware had already checked whether there was such a transaction, and if yes, would have aborted T_i. Hence, once a transaction is added to $tocommit\_queue\_k$, it will commit at R^k. Since it is the central middleware that makes the decision to add a transaction to

either all or none of $tocommit\_queue\_k$, the transaction will commit at all or none of the databases.

For property (iii), we now need to prove that the USG avoids G-1c cycles (i.e., cycles consisting entirely of read- or write-dependency edges) and G-SIb* cycles (i.e., cycles where each anti-dependency is prefixed by a read- or write-dependency edge).

(1). Assume G-1c exists. Note that all transactions in this cycle must be write transactions, since the source node of a read- or write-dependency edge must be a write transaction, and each node in the cycle is source node of one edge. Now consider an edge $T_i \xrightarrow{wr/ww} T_j$ in the cycle. If the edge is a read-dependency edge, it must appear in the SSG of T_j's local replica (assume R^l). Since all local histories are SI-histories, T_i must commit before T_j at R^l. Since SRP commits all write transactions in the same order at all replicas, T_i commits before T_j at all replicas (i.e, $c_i \prec_t c_j$). It will be the same situation if the edge is a write-dependency edge.

Hence, a G-1c cycle would result in $c_i \prec_t c_i$ at all local histories which is impossible. Hence, there is no such cycle.

(2). Assume G-SIb* exists. We can break the cycle into m sections of

$T_{i_p} \xrightarrow{(wr/ww)^*} T_{j_p} \xrightarrow{wr/ww} T_{k_p} \xrightarrow{rw} T_{i_{(p+1)\%m}}$ (where $0 \leq p < m$)

In section p, T_{i_p}, T_{j_p}, and $T_{i_{(p+1)\%m}}$ must be write transactions. $T_{k_p} \xrightarrow{rw} T_{i_{(p+1)\%m}}$ must be caused by the read operation(s) of T_{k_p}. Since SRP is ROWA, the read operations can only happen at T_{k_p}'s local replica (R^l). Hence, this edge occurs in $SSG(RH^l)$.

$T_{j_p} \xrightarrow{wr/ww} T_{k_p}$ is either a read- or a write-dependency edge. Let's consider the first case, i.e., it is a read-dependency edge. The edge must be caused by the read operation(s) of T_{k_p}. The edge must appear at T_{k_p}'s local replica R^l. We have already shown that this implies $c_{j_p} \prec_t s_{k_p} \prec_t c_{i_{(p+1)\%m}} \Longrightarrow c_{j_p} \prec_t c_{i_{(p+1)\%m}}$, which means that T_{j_p} commits before $T_{i_{(p+1)\%m}}$ at T_{k_p}'s local replica R^l.

Let's consider the second case, i.e., $T_{j_p} \xrightarrow{ww} T_{k_p}$. According to ROWA, the edge must appear in all replicas' SSGs, including T_{k_p}'s local replica R^l. We have shown that $T_{k_p} \xrightarrow{rw} T_{i_{(p+1)\%m}}$ also appears in SSG(RH^l). With the same reasoning as in the first case, we know T_{j_p} must commit before $T_{i_{(p+1)\%m}}$ at T_{k_p}'s local replica R^l (i.e., $c_{j_p} \prec_t c_{i_{(p+1)\%m}}$ at R^l).

Now let's consider $T_{i_p} \xrightarrow{(wr/ww)^*} T_{j_p}$. Obviously this implies $c_{i_p} \prec_t c_{j_p}$. Since T_{i_p} and T_{j_p} are write transactions, they must commit in the same order at all replicas (including R^l) according to

SRP. Hence, we derive $c_{i_p} \prec_t c_{i_{(p+1)\%m}}$ at R^l for section p. Let's put all sections together. We derive $c_{i_0} \prec_t c_{i_1} \prec_t ... \prec_t c_{i_{m-1}} \prec_t c_{i_0}$ in RH^l which is impossible.

Hence, G-SIb* can not happen in the $USG(RH)$ of a replicated history RH produced by SRP. □

4.2 Problems of SRP due to first-updater-wins strategy

SRP is a middleware-based approach. Hence, it has to work properly with the transaction processing mechanisms of the underlying database system. SRP works fine with SI databases that detect conflicts only at transaction commit time according to the first-committer-wins rule. However, real databases supporting SI are typically implemented with the first-updater-wins rule, as explained in Section 3.1.

Let's have a careful look at such an implementation. Before a transaction T_i updates a data item x it gets an exclusive lock on x. Once it has the lock, it performs a version check. If the latest committed version is from a concurrent transaction, T_i immediately aborts, otherwise it continues. If another transaction T_j holds a lock on x when T_i is requesting it, T_i has to wait until T_j terminates. If T_j terminates, T_i will immediately be aborted (since there is a committed version created by T_j that is concurrent to T_i). If T_j aborts, then T_i gets the lock but still performs the version check (because there could still be another concurrent transaction T_k that updated x before T_j but committed). This means, validation is not done at the end of transaction but on a continuous basis, namely always before an update operation is executed.

4.2.1 Blocking

Using the first-update-wins rule, transactions can block while waiting for locks to be released. This can lead to some problems using SRP.

Firstly, remote transactions might be blocked by local transactions in a database replica. Assume a transaction T_i executing locally at R^k and holding a lock on x. A remote transaction T_j has been validated and now is the first in $tocommit\_queue\_k$. T_j's writeset is applied at R^k and also updates

x. T_j has to acquire a lock and will be blocked. Ideally, T_i should be aborted since it conflicts with T_j and has not yet been validated. The middleware will detect this conflict once T_i finishes execution and validates since $T_i.start < T_j.tid$. Hence, T_i fails validation and aborts. At this point, T_j receives the lock and can execute the write. It will not abort since T_i aborted.

Secondly, it is possible to have deadlocks between local transactions that have not yet finished execution, and remote transactions that apply their writesets. The database detects such deadlock and aborts any of the transactions. If the local transaction is aborted, the middleware can simply inform the client (as is usually done with aborts due to deadlock). If the remote transaction is aborted, the middleware has to reapply the writeset until the remote transaction succeeds. We refer to this as *Adjustment 1*. Note that so far it is impossible that a local transaction that has completed execution and is validated is involved in such a deadlock because this local transaction has already acquired all necessary locks at the local database replica. Once it is validated, no further operations except for the commit are performed at the local replica.

Adjustment 1: To solve the blocking problems under the first-updater-wins rule

Upon T_i is first in $tocommit\_queue\_k$,

if T_i is remote at R^k, loop

- begin T_i^k at R^k
- apply $T_i.WS$ to R^k
- if T_i aborted by deadlock, then
  - continue the loop
- else (successfully applying $T_i.WS$)
  - same as SRP
  - exit the loop

4.2.2 Distributed deadlock

The third problem due to the first-update-wins strategy is the most serious problem. SRP might have a deadlock involving a cycle across the middleware and the database. Let's look at an example.

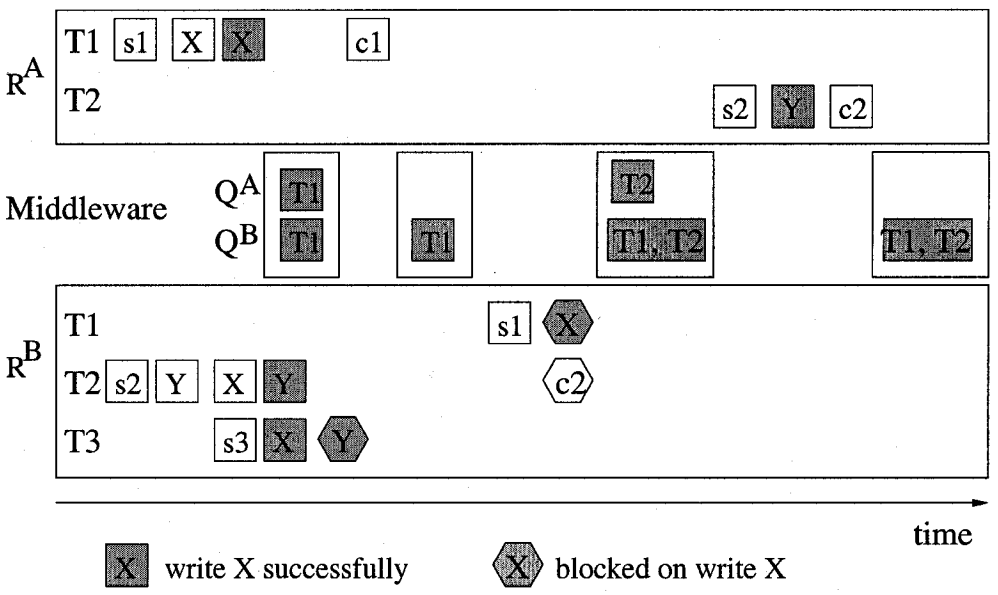


Figure 4.3: SRP Deadlock example execution with real databases

Example 12. *We extend Example 11 by adding one more operation $w_3(y)$ to T_3. We rearrange the interleaved order of operations as in Figure 4.3. We use polygon boxes to represent actions being submitted but blocked. White boxes are for read, start, commit, and abort operations, and grey boxes for write operations.*

T_1 starts at R^A. It reads and updates x. As in Figure 4.2, upon T_1's commit request, validation succeeds and T_1 receives $T_1.tid = 1$. T_1 is appended to Q^A and Q^B. T_1 commits at R^A and is removed from Q^A.

At R^B, T_1's writeset can be applied since T_1 is also the first in Q^B. But two local transactions have already performed operations at R^B. T_2 has read x and y, and updated y. Since the database uses locking for writes, T_2 has a lock on y. T_3 has updated x and holds a lock on x. T_3 is now blocked on y since it wants to write y but T_2 has a lock on y. Upon T_2's commit request, T_2's validation succeeds and it is appended to Q^A and Q^B.

At R^A, since T_1 has already been committed and removed from Q^A, T_2 is the first in Q^A. Hence, T_2's writeset is successfully applied and T_2 commits at R^A.

At R^B, T_2 is not the first in Q^B so it has to wait for T_1 to be committed. However, at R^B, T_1 is blocked by T_3 since it needs a lock on x which is held by T_3. T_3 is in turn blocked by T_2. There

is a deadlock among T_1, T_2 and T_3 at R^B (i.e., T_1 waits for T_3, T_3 waits for T_2, and T_2 waits for T_1). Note that there is no deadlock in the database. Only because T_2 waits for T_1 at the middleware layer, there is a distributed deadlock spanning both the middleware and the database.

The deadlock problem in the example above is due to the fact that T_1 must commit before T_2 even though they do not have write/write conflicts. In order to solve this distributed deadlock problem, we have two options. Firstly, we could avoid such deadlocks by allowing T_2 to commit before T_1. At the first view, this should be fine since they do not conflict. However, simply allowing transactions to commit out of validation order could lead to the violation of 1-copy-SI. Recall that Example 3 in Section 3.2.3 shows a replication history RH_{hole} that is not 1-copy-SI due to the fact that two update transactions commit in different order at two replicas. $USG(RH_{hole})$ has a G-SIb* cycle (i.e., a cycle in which each anti-dependency edge is prefixed with a start-dependency edge). Thus, a protocol that allows for out-of-order commits has to be carefully designed to avoid such phenomenon.

The second option is to detect such deadlocks, e.g., simply by using a timeout and resolve it by aborting one transaction. Note that such a distributed deadlock always involves a local transaction T that has finished execution and has validated but waits for a transaction T' in the $tocommit\_queue\_k$ to terminate.

The question now is which transaction to abort. Aborting T' will not help because we have to reapply it again before T and thus, the deadlock will again occur. Therefore, we abort T, that is, we abort a local transaction that has already been validated and has been waiting in the queue longer than a predefined threshold value. In above example, we abort T_2. Thus, the local transaction T_3 gets its lock and the deadlock is broken. However, this approach requires us to reexecute T at its local replica after T' has finished (since in principle, T should succeed). We do this by simply taking its writeset and apply it as we do at remote replicas. Reapplying the writeset has been done in Adjustment 1 so we can just reuse it.

We refer to the procedure as *Adjustment 2*.

Adjustment 2: To solve deadlock using timeout

- Add to step 1(d)iv in SRP (Upon transaction T_i successfully validates),
  - $\forall R^k : T_i.timeout := current\_time$
- Upon timeout of transaction T_i local at R^k,

  (i.e., $current\_time\text{-}T_i.timeout > timeout\_threshold$),
  - abort T_i at R^k
  - $T_i.aborted$:=true
- Upon T_i is first in $tocommit\_queue\_k$,

  if T_i is remote at R^k or $T_i.aborted$=true, loop
  - begin T_i^k at R^k
  - apply $T_i.WS$ to R^k
  - if T_i aborted by deadlock, then
    - continue the loop
  - else (successfully applying $T_i.WS$)
    - same as SRP
    - exit the loop

Discussion:

Note that the distributed deadlock only spans across the middleware and the underlying database at one replica. It does not involve any interaction between different replicas, which make it easy to detect and resolve. [82] suggests to detect the deadlock by querying the lock tables provided by DBMSs, such as the pg_locks view of the PostgreSQL system catalogue, and similar tables or views in other databases [82].

4.3 Problems of SRP due to integrity constraints

SRP is based on the assumption that there are no integrity constraints in the database. Hence, it only guarantees 1-copy-SI and does not work correctly if integrity constraints are considered. At this

timepoint, it is important to understand how databases that provide SI guarantee IC-consistency. We have analyzed PostgreSQL. In here, integrity reads do not read from a snapshot. Instead, a transaction T_i performing an integrity read on x acquires a read lock on x. If an exclusive lock is set by a transaction T_j writing x (first-updater-wins strategy), T_i has to wait until this lock is released. If T_j commits, T_i reads T_j's version. This guarantees that T_i reads the latest committed version. If the read determines that a constraint would be violated, T_i aborts. Otherwise, T_i continues to write and then commit. T_i keeps the lock on x until termination. If a further transaction T_k wants to update x it is blocked by T_i. This guarantees that nobody overrides the value of x and commits before T_i commits, a requirement for IC-consistency. At the end, the SSG will have an IC-read-dependency edge and a commit-dependency edge from T_j to T_i and and IC-anti-dependency edge and a commit-dependency edge from T_i to T_k (assuming that T_k is not concurrent with T_j). That is, integrity reads basically set long read locks just as in strict 2PL in order to guarantee IC-consistency.

Example 13. *Now let's look at an Example similar to Example 5 but in a replicated environment. There are tables* Dept(did, dname) *and* Emp(eid, ename, did). *A department record* ('d1', 'marketing'), *referred to as object x, exists, inserted by transaction T_0. Now assume a transaction T_1 inserts an employee* ('e1', 'Mike', 'd1'), *denoted as y, and a transaction T_2 deletes the department. We assume there are two replicas R^A and R^B. T_1 is submitted to R^A and T_2 is submitted to R^B. Let's look at a possible execution in SRP. Execution is also depicted in Figure 4.4.*

At R^A, T_1 performs an integrity read $ir_1(x_0)$ and then a write $w_1(y_1)$. At R^B, T_2 performs an integrity read $ir_2(y_{init})$ and then a write $w_2(x_{dead})$. Assume T_1 finishes first. The middleware validates and puts T_1 in both queues. Then T_2 finishes. The middleware validates T_2 and appends it to both queues. At R^A, T_1 can commit. However, when the writeset of T_2 is applied, the database replica will perform the integrity read, find the employee tuple y_1, and abort T_2. Given our Adjustment 1 *of Section 4.1.1, we will attempt to apply the writeset of T_2 over and over again and run into an endless loop.*

At R^B, when T_1 is applied, it will perform its integrity read, acquiring a read lock on x. T_1 will be blocked at the database replica since T_2 has an exclusive lock on x. T_2, in turn, is waiting behind T_1 in Q_B waiting for T_1 to finish. Thus, we have a distributed deadlock between the middleware and

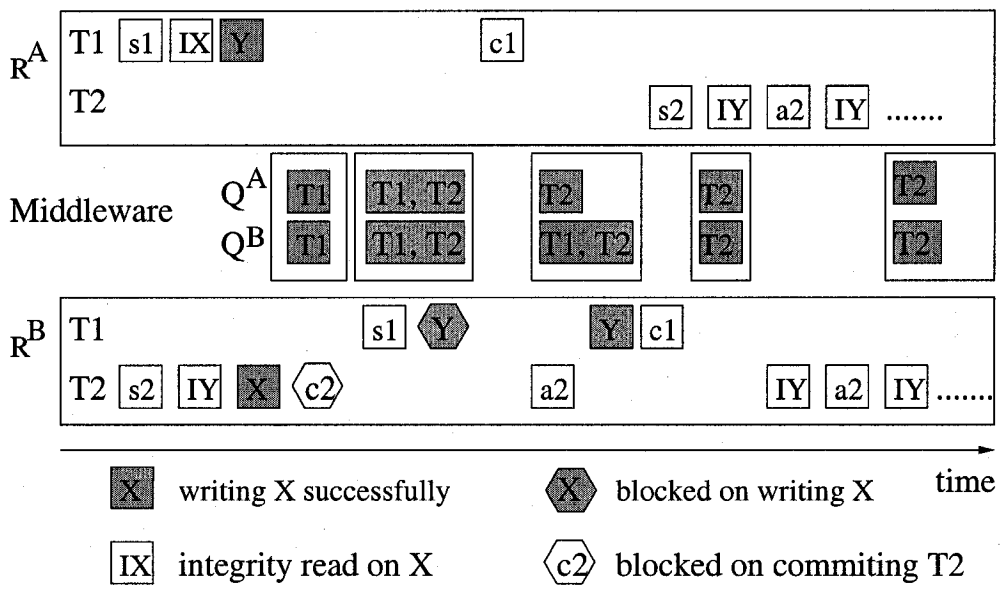


Figure 4.4: Example with foreign key constraints executed according to SRP with Adjustments 1 and 2.

the database replica. With our Adjustment 2 in Section 4.2.2, T_2 will be aborted due to a timeout. However, when T_2 is now reapplied, it's integrity read will now read y_1, and thus, T_2 will abort due to integrity violation. We will again run into an endless loop.

Actually, many of the update everywhere approaches [6, 69, 91, 92, 90, 61, 73, 124] can run into problems with integrity constraints, no matter if they provide 1-copy-SE or 1-copy-SI.

The issue is that a transaction T, while executing locally, performs integrity reads that do not indicate any violation of integrity constraints. However, T might then be validated by the middleware after a remote transaction T' whose write operations actually lead to a violation. When T's writes are applied, the database replica automatically reevaluates the integrity constraints through integrity reads, detects a violation and aborts T. The problem is that the middleware has no means to check the integrity constraints since it is not aware of integrity constraints. Considering only foreign keys, such information could be extracted, e.g., by looking at the database schema, but in the general case, this is not possible.

Our solution is as follows. We do not check integrity constraints at the middleware but let the database do it (i.e., perform integrity constraints). The middleware only checks for write/write

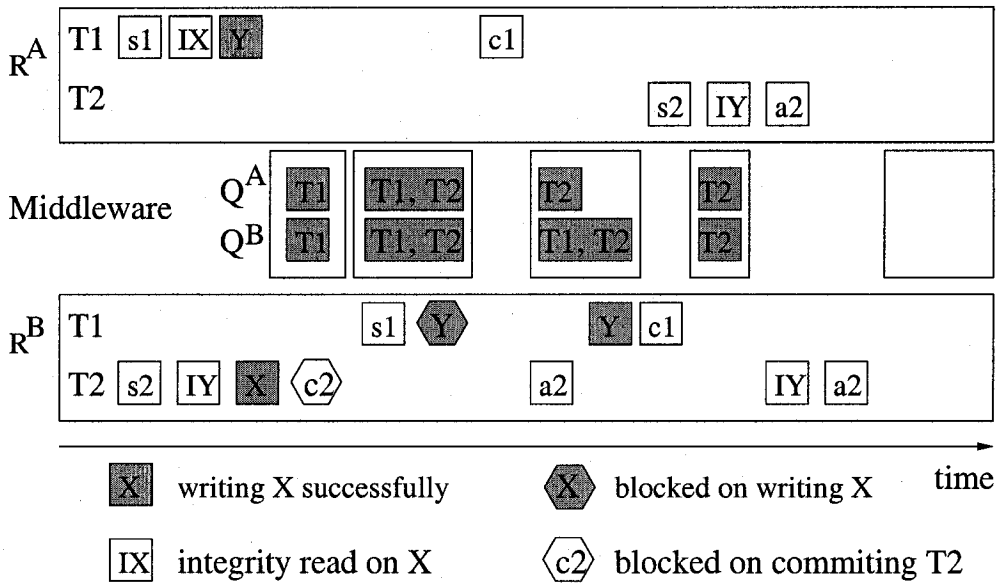


Figure 4.5: Revisit Example 13 with Adjustment 3

conflicts as before. We make sure that at each database replica, integrity reads read the same data versions, as has been done in the example above. Thus, either at all replicas the integrity reads will determine a violation of the constraints and abort, or the write operations will succeed.

When using this approach we have to be aware that a transaction might abort for several reasons. For example, in Section 4.2.1, we have seen that a database replica might abort a transaction because of a deadlock. Thus, we must be able to determine the reason for an abort and then act appropriately. Fortunately, using a typical database interfaces such as JDBC, if a transaction aborts, the database returns an error message and a SQLSTATE code that indicates the reason for the abort. Using the code, we can decide whether an abort was due to a deadlock or an integrity constraint.

Example 14. *Let's now revisit Example 13 with the idea above. The new execution scenario is in Figure 4.5, when transaction T_2 aborts at R^A, the error message will indicate that this abort is due to an integrity violation. Thus, we do not reapply T_2. At R^B, the first abort of T_2 is induced by the middleware. The middleware now reapplies T_2 after T_1 commits. This time, T_2 is aborted by the database replica because of integrity violation. The middleware will detect this by looking at the abort exception and not reapply T_2. At the end, T_1 commits and T_2 aborts in both replicas. The history is 1-copy-SI+IC.*

In the following, we combine our above solution to integrity constraints with Adjustment 1 and refer to this as *Adjustment 3*. We use bold letters to highlight the difference to Adjustment 1.

Adjustment 3: To handle deadlocks due to integrity constraints and solve blocking problems due to first-updater-wins

Upon T_i is first in $tocommit\_queue\_k$,

if T_i is remote at R^k or $T_i.aborted$=true, loop

- begin T_i^k at R^k
- apply $T_i.WS$ to R^k
- if T_i aborted by deadlock, then
  - continue the loop
- **else if T_i aborted by IC, then,**
  - **exit the loop**
- else (successfully applying $T_i.WS$)
  - same as SRP
  - exit the loop

4.4 Simple Replication Protocol with Integrity Constraints (SRP-IC)

The SRP-IC protocols extend SRP by integrating the Adjustments 2 and 3. That is, SRP-IC has to consider aborts due to deadlocks within the database and integrity constraint violations, and implements its own timeout mechanism in order to handle deadlocks distributed across the middleware and the database.

4.4.1 Protocol details

Figure 4.6 provides the full description of SRP-IC. Since SRP-IC is mainly based on SRP, we highlight the adjustments with bold letters for the convenience of reading.

Additionally to what we have indicated in Adjustment 2 and 3 there is an additional $timeout\_mutex$ variable for each transaction. After a transaction T_i local at R^k is validated at the middleware, its $timeout$ variable is set to the current time (step 1(d)iv). Once the difference between the $timeout$ value and the current time becomes larger than a predefined threshold and the transaction is still not the first in $tocommit\_queue\_k$, T_i is aborted (step 3). The transaction is marked as aborted so when it becomes the first in the queue, we know that its writeset has to be applied (usually the writesets of local transactions do not need to be applied).

Another major change is in handling the first transaction in the $tocommit\_queue\_k$. Once the transaction becomes the first in the queue, if it is local at R^k, its $timeout$ variable is set to 0, indicating that the transaction should not be aborted (step 2a) anymore. Note that checking the $timeout$ variable in step 3, and resetting it to 0 in step 2a are done using the $timeout\_mutex$ so that a transaction is not aborted at the same time it is trying to commit. Thus, steps 1(d)iv, step 3, and step 2a are needed for Adjustment 2. Applying a writeset is now needed for remote transactions and for local transactions that got aborted.

Furthermore, when applying a writeset, the transaction might get aborted due to a database internal deadlock or due to integrity constraints. If the abort is due to a deadlock, we simply reapply (step 2(b)iii) as required by Adjustment 2. If T_i is aborted due to an integrity constraint (step 2(b)iv), we do not reapply according to Adjustment 3. However, we increase $lastcommitted\_tid\_k$ in order to keep it synchronous with $next\_tid$ (step 2(b)ivC). The remainder of step 2 is the

same as for SRP. Once T_i is successfully applied (step 2(b)ivB), we can commit T_i and increase $lastcommitted\_tid\_k$ (step 2(b)ivC).

In any case (i.e., committed or aborted), T_i will be removed from the $tocommit\_queue\_k$ (step 2d).

4.4.2 Correctness

Theorem 7. *SRP-IC provides 1-copy-SI+IC if the underlying database replicas provide SI+IC using the first-updater-wins strategy.*

Proof. The proof is similar to the proof of Theorem 6.

Based on Theorem 5, we need to show that for any replicated history RH possible under SRP-IC, (i) the local histories RH^k at all replicas are SI+IC histories, (ii) an update transaction commits at either none or all replicas, and (iii) there exists a $USG\text{-}IC(RH)$ that has no G-1c* and G-SIb* cycles.

Property (i) is fulfilled since the underlying database replicas provide SI+IC by assumption. For property (ii) we need to show that, apart of read-only transactions, all replicas commit the same set of transactions. If validation of a transaction T_i succeeds at the middleware it is appended to the $tocommit\_queue\_k$ of each replica R^k. Thus, a transaction T_i is either in all or none of the queues and the order of transactions in all queues is the same. Transactions in each $tocommit\_queue\_k$ are handled one after the other. That is, they are committed/aborted in the same order at all replicas. We now show by induction on the position of the transaction that all database replicas will decide the same outcome for each individual transaction.

We show first that all replicas will commit the first transaction T_1 validated. The transaction is assigned a *tid*:=1. At T_1's local replica R^l, when T_1 is put in $tocommit\_queue\_l$, it is the first in the queue. Thus, T_1^l simply commits. At a remote replica R^k a transaction T_1^k is started to apply the writeset. This includes the integrity reads related to the write operations. T_1^k might be blocked by local transactions or even abort (due to deadlock) and be restarted. Nevertheless, once it performs the integrity reads it will read the same versions as T_1^l has done at the local replica R^l because no transaction will commit after T_1^k starts, and thus, no violation will be determined and T_1^k will

Initialization: $next\_tid := 1$, $ws\_list := \{\}$
$\forall\ R^k$: $tocommit\_queue_k := \{\}$
$\forall\ R^k$: $lastcommitted\_tid\_k := 0$
$wsmutex$,
$\forall\ R^k$: $dbmutex\_k$
$\boldsymbol{timeout\_threshold := certain\_value}$ **(e.g., 50ms)**

1. Upon receiving an operation Op_i of T_i,
   (a) if Op_i is start (i.e.,s_i), then
      i. choose R^k at which T_i will be local
      ii. obtain $dbmutex\_k$
      iii. $T_i.start := lastcommitted\_tid\_k$
      iv. begin T_i^k at R^k
      v. release $dbmutex\_k$
      vi. return to client
   (b) else if Op_i is read or write, then
      i. execute at local R^k and return to client
   (c) else if Op_i is abort, then
      i. abort T_i^k at R^k and retun to client
   (d) else (commit)
      i. $T_i.WS := getwriteset(T_i^k)$ from local R^k
      ii. if $T_i.WS = \emptyset$, then
         • commit and return
      iii. obtain $wsmutex$
      iv. if $\nexists T_j \in ws\_list$ such that $T_i.start < T_j.tid \wedge T_i.WS \cap T_j.WS \neq \emptyset$:
         • $T_i.tid := next\_tid++$
         • append T_i to ws_list
         • $\boldsymbol{\forall R^k}$**:** $\boldsymbol{T_i.timeout}$**:=**$\boldsymbol{current\_time}$
         • $\forall R^k$: append T_i to $tocommit\_queue\_k$
         • release $wsmutex$
      v. else (validation fails),
         • release $wsmutex$
         • abort T_i^k at R^k

2. Upon T_i is first in $tocommit\_queue\_k$.
   (a) **if T_i local at R^k,**
      i. **obtain $T_i.timeout\_mutex$**
      ii. $\boldsymbol{T_i.timeout := 0}$
      iii. **release $T_i.timeout\_mutex$**
   (b) **if T_i remote at R^k or $T_i.aborted$=true, loop**
      i. begin T_i^k at R^k
      ii. apply $T_i.WS$ to R^k
      iii. **if T_i aborted by deadlock, then**
         • **continue the loop**
      iv. **else (if T_i aborted by IC or successfully applied)**
         A. obtain $dbmutex\_k$
         B. **if T_i successfully applied**
            • commit T_i at R^k
         C. lastcommitted_tid_k++
         D. release $dbmutex\_k$
         E. **exit the loop**
   (c) if T_i is local at R^k, return to client
   (d) remove T_i from $tocommit\_queue\_k$

3. **Upon a transaction T_i local at R^k timeout, (i.e., $current\_time$ - $T_i.timeout > timeout\_threshold$),**
   (a) **obtain $T_i.timeout\_mutex$**
   (b) **if $T_i.timeout \neq 0$, then**
      i. **abort T_i at R^k**
      ii. $\boldsymbol{T_i.aborted}$**:=true**
   (c) **release $T_i.timeout\_mutex$**

Figure 4.6: SRP-IC: a Simple Replication Protocol with Integrity Constraints

eventually commit.

We now assume the $n(n \geq 1)$ transactions have been validated and all replicas have made the same commit/abort decision for these transactions.

Now let's have a look at the next validated transaction T_{n+1}. For any remote replica R^k, when T_{n+1} is the first in *tocommit_queue_k*, all transactions T_1 to T_n have terminated and it is guaranteed that no further transaction will commit until T_{n+1} commits or aborts due to integrity constraints. Thus, T_{n+1} at each R^r (R^r being a remote replica of T_{n+1}), will do its integrity reads exactly on the same data versions, and thus either all remote replicas will detect an integrity violation and abort the transaction, or will not detect an integrity violation and commit the transaction eventually (potentially after a sequence of aborts due to deadlock). We now have to show that T_{n+1} performs exactly the same integrity reads at the local replica R^l. We distinguish two cases. First, assume when T_{n+1} is the first in the queue it was already aborted due to a timeout (step 3 of the protocol). In this case T^l_{n+1} is restarted just as it were a remote transaction. That is when T^l_{n+1} is restarted at R^l, all transactions T_1 to T_n have terminated at R^l, and thus the new T^l_1 will perform the same integrity reads as at the remote replicas, and thus, make the same decision.

Now assume T^l_{n+1} was not yet aborted when it is the first in the queue. Let T^l_{n+1} have performed an integrity read $ir^l_{n+1}(x_i)$. We have to show that T_i was the last transaction in the sequence $T_1, ...T_n$ to have written x and commit. Then we can be sure that all remote transactions at remote replicas R^r perform the same integrity read $ir^r_{n+1}(x_i)$ since $ir^r_{n+1}(x_i)$ means that T_i was the last to write x and commit before the read occurred. At R^l, before performing the integrity read T^l_{n+1} acquired a lock on x. At this time T_i was the last to write x and commit. T^l_{n+1} keeps the lock until it terminates. Now assume a transaction T^l_j, $i < j \leq n$ updated x, i.e., T_i is not the last in the sequence to update x. However, when T^l_j requests a lock on x at R^l it is blocked on T^l_{n+1}. Since T^l_{n+1} is after T^l_j in the *tocommit_queue_l*, there is a deadlock. T^l_j waits for T^l_{n+1} in the database to release the lock on x, T^l_{n+1} waits for T^l_j in the queue to terminate. We have a distributed deadlock. According to the protocol, T^l_{n+1} is aborted. Thus, our assumption does not hold that T^l_{n+1} was not yet aborted when it is the first in *tocommit_queue_l*. Thus, if it is the first in the queue and was not yet aborted and it has performed $ir^l_{n+1}(x_i)$, we can be sure that T_i was the last transaction in the sequence $T_1, ...T_n$ to update x and commit. Therefore, at all remote replicas R^r, T^r_{n+1} will perform

the same $ir^r_{n+1}(x_i)$ and decide a commit.

For property (iii), we need to show that there exists a $USG\text{-}IC(RH)$ avoiding G-1c* cycles (i.e., cycles consisting entirely of read-, write-, IC-read-, or IC-anti-dependency edges) and G-SIb* cycles (i.e., cycles where each anti-dependency edge is prefixed by a read- or write-dependency edge). We have just shown in our proof of property (ii) that all replicas commit the same set of transactions in exactly the same order, and that all committed transactions perform their integrity reads on exactly the same data versions. This means, for any two replicas R^k and R^l, if there is an IC-read- or IC-anti-dependency edge from T_i to T_j in $SSG(RH^k)$, then there is the same edge from T_i to T_j in $SSG(RH^l)$. As a result, there exists actually only a single $USG\text{-}IC(RH)$, since independently which replica R^k we choose for a transaction T_i, its IC-dependency edges are the same as in other replicas. We now show that this $USG\text{-}IC(RH)$ avoids G-1c* and G-SIb* cycles.

Assume a G-1c* cycle exists in $USG\text{-}IC(RH)$. There can be four kind of edges in the cycle: read-, write-, IC-read-, and IC-anti-dependency edges. Note that all transactions in the cycle must be write transactions. This is true because each transaction in the cycle is the start node of a read-, write-, IC-read, or IC-anti-dependency edge. If it is the start node of a read-, write-, or IC-read-dependency edge it is obviously an update transaction. Being the start node of a IC-anti-dependency edge means the transaction performed an integrity read which is followed by a successful write operation. Thus, all transactions are update transactions, and thus, are executed at all replicas. Each edge in the cycle occurs at least in the $SSG(RH^k)$ of one local history RH^k, and since RH^k is a SI+IC history implies $c_i \prec_t c_j$ in this history. Since all histories commit write transactions in the same order, this also implies $c_i \prec_t c_j$ in all other local histories. Therefore, the G-1c* cycle in $USG\text{-}IC(RH)$ implies $c_i \prec_t c_i$ in the local histories, which is impossible.

The proof that no G-SIb* cycle is similar to the proof for correctness of SRP (Theorem 6) and omitted here. □

4.5 Discussion

This chapter presents two protocols, SRP and SRP-IC. They both have the central architecture as in Figure 2.1.(a) without considering message overhead and fault-tolerance.

Protocols	architecture	isolation guarantee	works for databases with
SRP	centralized (Fig 2.1.(a))	1-copy-SI	first-committer-wins
SRP-IC	centralized (Fig 2.1.(a))	1-copy-SI+IC	first-updater-wins

Table 4.1: Comparison of SRP and SRP-IC

SRP addresses the practicability problems of existing protocols. Transactions do not need to provide information at start time. However, it only guarantees 1-copy-SI. Moreover, it does not work for database systems with the first-updater-wins rule, where deadlocks can occur. SRP-IC extends SRP to work with databases using the first-updater-wins rule and provides 1-copy-SI+IC. It handles the deadlock problem by using a timeout mechanism.

We summarize the differences between SRP and SRP-IC in Table 4.1.

Chapter 5

Replica control for performance and fault-tolerance

In the last chapter we developed SRP-IC. It provides 1-copy-SI+IC for database systems implementing SI with the first-update-wins strategy. However, the protocol ignores important issues such as performance and fault-tolerance. In the following, Section 5.1 discusses these problems carefully. Then, Section 5.2 proposes a new protocol, which we call *Snapshot Isolation based on MultiCast (SIMC)*, that addresses the problems. SIMC is based on multicast primitives provided by group communication systems (GCS). However, our analysis shows that these multicast primitives are not good in wide area networks (WANs). Hence, Section 5.3 develops a protocol, SEQ, that does not rely on group communication systems but integrates communication more tightly with replica control. However, SEQ has weaker fault-tolerance than SIMC. Thus, Section 5.4 combines SIMC and SEQ into a new protocol, HYBRID, that takes advantage of network topologies. Its performance and fault-tolerance guarantees lie in between those of SIMC and SEQ.

5.1 Problems of performance and fault-tolerance in SRP-IC

While SRP-IC is likely to work well in a LAN it will not in a WAN. The reason is the centralized architecture (Figure 2.1.(a)) used in the protocols. Since there is only one middleware component,

all requests must go through it. This results in WAN communication between the middleware and clients if the clients are remote, and between the middleware and the database replicas if the database replicas are distributed across the WAN. Communication is necessary for each read and write operation. Recall that there might be more than one operation in one transaction. Thus, the response time of a transaction will include the time needed for several message rounds across the WAN.

Regarding fault-tolerance, clearly the single middleware is a single point of failure. Having a single backup will provide fault-tolerance but is complicated, as discussed in Section 2.2.2. Furthermore, it does not help to handle the performance problem in a WAN.

Actually, many lazy primary approaches and update everywhere approaches with a central scheduler experience the same problems as SRP-IC, since they follow the centralized architecture. They have proven to work fairly well in LANs, but not in WANs.

To reduce the WAN communication overhead, especially the one that occurs within the response time of a transaction, we should keep the number of WAN messages as low as possible. Section 2.2.7 gave an overview of existing replication protocols based on GCS. They can be categorized into two categories according to when multicast is used, i.e., before or after transaction execution. In these protocols only one single multicast message is needed within the response time of an update transaction. All approaches are either kernel-based or use the decentralized middleware architecture (Figure 2.1.(c)) in which there is one middleware instance for each database replica. Having a single message round is, in principle, good for performance. The reliablity guarantees are good for fault-tolerance. Thus, this chapter explores how the properties of GCS can be used for replication protocols providing 1-copy-SI+IC.

5.2 SIMC: a replication protocol based on Group Communication Systems (GCS)

5.2.1 Basic idea

In this section we propose a protocol SIMC that guarantees 1-copy-SI+IC with the cost of one multicast message per update transaction. It extends SRP-IC and assumes databases using the first-updater-wins strategy. SIMC uses the decentralized architecture shown in Figure 2.1.(c). Each site has a middleware replica connecting to a local database replica. Clients submit their transactions to one of the middleware replicas. In a WAN, this will be the one closest to the client.

As in SRP-IC, a transaction T is executed optimistically in the local database replica. Receiving the commit request from the client, the middleware retrieves the writeset from its local database. Recall that SRP-IC depends on the single middleware component to make a unique decision of commit/abort. Since there are several middleware replicas in SIMC, the middleware replicas need to synchronize in order to make an unique decision to commit or abort the transaction.

SIMC uses a GCS for communication among the middleware replica, and depends on the total order multicast provided by GCS to guarantee that a unique decision is made. Recall that total order multicast guarantees that all sites receive messages m_1 and m_2 in the same order. SIMC multicasts the writesets of transactions in total order. Hence, all middleware replicas receive the writesets in the same order. As long as all middleware replicas perform validation for these transactions in their delivery order, the decision will be the same at all replicas. The validation itself will be the same as in SRP-IC. We only need to check if two transactions are concurrent and have write/write conflicts.

Integrity constraints are handled in the same way as in SRP-IC. Each middleware replica applies remote transactions and commits all transactions in their corresponding local database replicas according to the order of validation. The databases will finally check integrity constraints and determine if an update transaction commits or aborts. The transaction will commit in either all or none of the replicas since it is applied according to the validation order. Deadlocks are again handled via timeouts. We defer the discussion of fault-tolerance to Section 5.2.6.

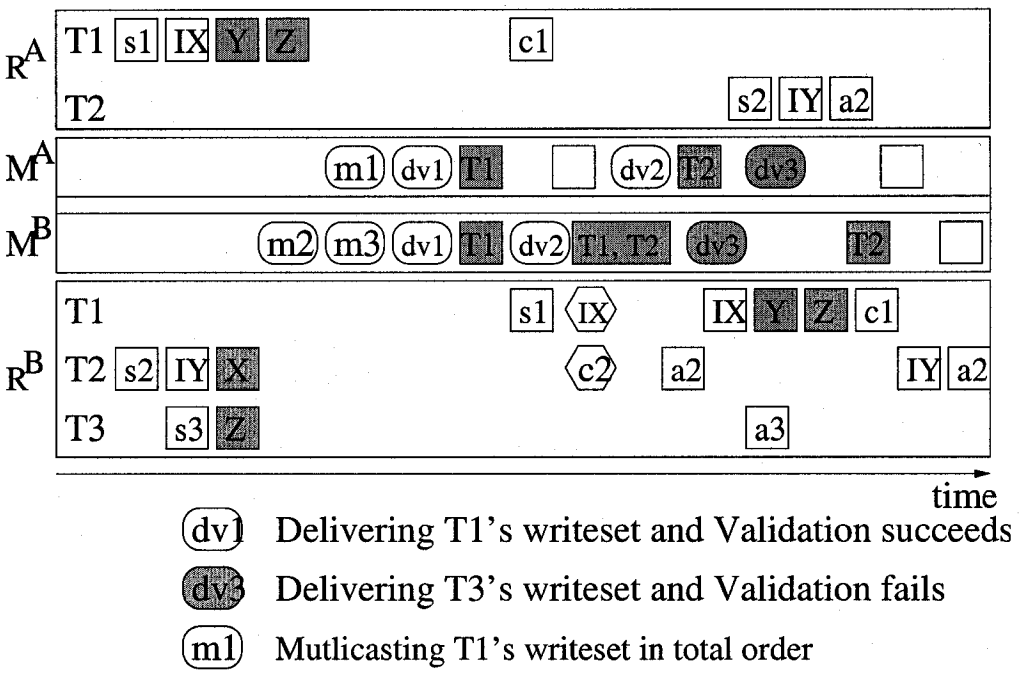


Figure 5.1: A SIMC example extended from Example 14 in Figure 4.5

5.2.2 Example

Example 15. *Let's consider an example extended from Example 14 shown in Figure 4.5 for SRP-IC.* T_1 *performs an integrity read on* x*, a write on* y*, and additionally a write on* z*.* T_2 *performs an integrity read on* y*, and a write on* x*. And additional transaction* T_3 *performs a write on* z*.* T_1 *is submitted to* R^A*.* T_2 *and* T_3 *are submitted to* R^B*. The execution scenario is shown in Figure 5.1. Note that we use ellipses to represent the multicast of a writeset and the validation of the transaction after the writeset delivery. We also separate the middleware replicas at site A and B.*

At replica A, T_1 *is the only local transaction and can finish its execution locally. The middleware replica* M^A *retrieves the writeset of* T_1 *and then multicasts it in total order. At the same time,* T_2 *and* T_3 *execute locally at* R^B*. They both succeed in their local execution since they do not block each other.* M^B *multicasts the writesets of* T_2 *and* T_3*. The writeset delivery order is* T_1 *then* T_2 *then* T_3 *at both replicas. Since* T_1 *is the first transaction to deliver and validate, validation succeeds at both replicas. So* T_1 *is inserted into the* $tocommit\_queue$ *at both* M^A *and* M^B*.*

At replica A, R^A *simply commits* T_1 *and* M^A *removes* T_1 *from its queue. Upon the delivery*

of T_2's writeset, M^A validates T_2 and applies the writeset immediately since T_2 is the first in the queue. The integrity read of T_2 finds a violation of integrity constraints. Hence, T_2 aborts and is removed from the queue. At the same time, M^A delivers the writeset of T_3. However, T_3's validation fails because T_3 has write/write conflicts with T_1, and T_3 and T_1 are concurrent. Hence, T_3 will be discarded by M^A.

At replica B, R^B applies T_1's writeset but the execution is blocked by the integrity read of T_2. Upon the delivery of T_2's writeset, M^B checks that T_1 has no write/write conflicts with T_2. Hence, the validation of T_2 succeeds and T_2 is inserted into the tocommit_queue. A deadlock occurs. Upon T_2's timeout, it aborts. Now, T_1 can apply its writeset and commit successfully. T_1 is removed from the queue and T_2 reapplies its writeset. But it will abort due to the integrity constraint. Upon the delivery of T_3's writeset, T_3 aborts because of T_1.

Finally at both replicas, T_1 commits but T_2 and T_3 abort. We would like to mention two points. The first is that read-only transactions do not need to be multicast and validated since they do not have writesets. The second is that the delivery and validation of a write transaction must be atomic, or at least validation must be performed according to the delivery order. Otherwise, the validation outcome will be different at different sites.

5.2.3 Protocol details

Figure 5.2 shows the details of SIMC. SIMC is very similar to SRP-IC shown in Figure 4.6. It is implemented at the middleware level and deployed in each middleware instance M^k. SIMC is different from SRP-IC only in total order multicast after local execution (step 1(d)iii), and in total order delivery (step 2). We highlight them in bold letters. Besides, note that there is one middleware replica M^k for each database R^k. There is one set of data structures (e.g., ws_list, $wsmutex$, $dbmutex$, $tocommit\_queue$, $lastcommitted\_tid$, $next\_tid$) for the middleware replica at each site.

A client of M^k submits the operations of its transactions only to M^k which executes them in R^k locally. A local transaction starts immediately when M^k receives its start operation (step 1a). Note that we keep track of the last committed transaction before T_i starts (step 1(a)ii). The subsequent read or write operations will be executed in R^k (step 1b). Upon the arrival of the commit request

Initialization:
lastcommitted_tid:=0, *next_tid*:=1
ws_list:={}, *tocommit_queue*:={}
wsmutex, *dbmutex*
timeout_threshold:=*certain_value* (e.g.,50ms)

1. Upon receiving an operation Op_i of T_i

(a) if Op_i is start, then
  i. obtain *dbmutex*
  ii. $T_i.start := lastcommitted\_tid$
  iii. begin T_i at R^k
  iv. release *dbmutex*
  v. return to client

(b) else if Op_i is read or write
  i. execute in local R^k and return to client

(c) else if Op_i is abort, then
  i. abort T_i at R^k and return to client

(d) else (commit),
  i. $T_i.WS := getwriteset(T_i)$ from local R^k
  ii. if $T_i.WS = \emptyset$, then
    • commit and return
  iii. **multicast the writeset of T_i in total order**

2. **Upon delivering T_i in total order**

(a) obtain *wsmutex*
(b) if $\nexists T_j \in ws\_list$ such that $T_i.start < T_j.tid \wedge T_i.WS \cap T_j.WS \neq \emptyset$
  i. $T_i.tid$:=*next_tid*++
  ii. append T_i to *ws_list*
  iii. $T_i.timeout$:= *current_time*
  iv. append T_i to *tocommit_queue*
  v. release *wsmutex*
(c) else
  i. release *wsmutex*
  ii. if T_i is local, then abort T_i at R^k and return to client

3. Upon T_i is first in *tocommit_queue*, then

(a) if T_i local at R^k,
  i. obtain $T_i.timeout\_mutex$
  ii. $T_i.timeout := 0$
  iii. release $T_i.timeout\_mutex$
(b) if T_i remote at R^k or $T_i.aborted$=true, loop
  i. begin T_i^k at R^k
  ii. apply $T_i.WS$ to R^k
  iii. if T_i aborted by deadlock, then
    • continue the loop
  iv. else (if T_i aborted by IC or successfully applied)
    A. obtain *dbmutex*
    B. if T_i successfully applied
      • commit T_i at R^k
    C. lastcommitted_tid++
    D. release *dbmutex*
    E. exit the loop
(c) return to client if T_i is local.
(d) remove T_i from *tocommit_queue*

4. Upon timeout of transaction T_i local at R^k, (i.e., *current_time* - $T_i.timeout >$ *timeout_threshold*)

(a) obtain $T_i.timeout\_mutex$
(b) if $T_i.timeout \neq 0$, then
  i. abort T_i
  ii. $T_i.aborted$:=true
(c) release $T_i.timeout\_mutex$

Figure 5.2: SIMC on M^k: a replication protocol based on total order multicast

(step 1d), M^k retrieves the writeset and checks if the writeset is empty or not. If it is empty, the transaction will be committed immediately. Otherwise, the writeset will be multicast in total order (step 1(d)iii).

While the writeset of T_i is delivered at a middleware replica M^k (step 2), M^k will validate the writeset transaction against all other transactions which have been validated successfully since T_i started (i.e., $tid > T_i.start$) (step 2b). If none of them has write/write conflicts with T_i, the validation succeeds and T_i is appended to the ws_list and $tocommit\_queue$. At the same time, $T_i.timeout$ is set to the current time at M^k. Otherwise, the validation fails and T_i is aborted if local or simply discarded (step 2c).

Once a validated transaction is the first in $tocommit\_queue$, it will be applied until success or until it aborted due to integrity constraints (step 3). A timeout mechanism is applied to solve the deadlock problems (step 4). These mechanisms are the same as in SRP-IC.

5.2.4 Correctness

In SRP-IC, there is a single middleware making the validation decision and appending transactions to queues. In SIMC, we have one middleware replica per database replica that performs validation. We have to show that all middleware replicas make the same decision on validation as the central middleware replica in SRP-IC. If we can show this, SIMC provides 1-copy-SI+IC.

We can show this by induction. The first transaction submitted to the replicated database always succeeds in its validation at all replicas since there is no transaction in ws_list at all replicas. The transaction is assigned a $tid := 1$ and the $next\_tid$ is set to 2.

We now assume that $n(n \geq 1)$ transactions validate successfully and reside in ws_list at all replicas. Each transaction has the same tid at all replicas, and the $next\_tid$ at any replica is set to (n+1). The next transaction T_{n+1} is multicast in total order and received by all replicas. At each replica, T_{n+1} is validated against all transactions in ws_list. Note that validation is performed according to transaction delivery order. Hence, T_{n+1} sees the same number of transactions in ws_list at all replicas. Recall that $T_{n+1}.start$ was assigned at T_{n+1}'s local replica so it is the same at any replica in validation. Thus, T_{n+1} validates successfully either in all or in none of the replicas.

Hence, all replicas make the same decision on validation. The remainer of the protocol is

basically the same as SRP-IC. Therefore, SIMC provides 1-copy-SI+IC.

5.2.5 An optimization: early validation

So far, we only check if an update transaction T_i is allowed to commit upon the delivery of its writeset. However, we can already perform a fair amount of validation earlier. For instance, we observe that at the end of the transaction, just before M^k multicasts T_i's writeset, M^k might have already received a concurrent transaction T_j having a write/write conflict with T_i. In the last example, before multicasting T_3's writeset, middleware replica M^B had already received and validated T_1's writeset. If M^B can detect this fact, it does not even need to multicast T_3's writeset because it is clear that T_3 will abort. This will reduce the response time of T_3 since T_3 can be aborted immediately. It also reduces network traffic. We refer to M^k validating before sending the writeset as *early validation*. In fact, during this early validation of a transaction T_i, M^k does not even need to validate against all transactions that are concurrent to T_i and have validated. Since the database system uses the first-update-wins strategy, part of the validation has actually already been done in the database replica.

Let's have a closer look at who should validate what. In principle, a transaction T_i, local at replica M/R^k, needs to be validated against all concurrent transactions that validated before T_i. We can categorize these concurrent transactions as follows. At the time M^k performs early validation (just before multicasting the writeset), (i) some of these concurrent transactions are already committed at R^k, and (ii) some have already arrived at M^k but are still residing in the to_commit_queue of M^k. Additionally, there are (iii) some transactions that will be delivered between the early validation at M^k and the time T_i's writeset is delivered.

For the transactions in category (i), that is, those concurrent transactions that have already committed locally at R^k, the database replica has actually already done the validation due to the first-update-wins strategy. According to this strategy, if a transaction is concurrent to T_i, has a write/write conflict, and commits, then T_i aborts when it attempts to perform the conflicting write operation. However, at the time of early validation, all of T_i's operations have executed, and T_i is not yet aborted. Hence, we can be sure that T_i does not conflict with any transaction that has already committed at R^k.

Transactions stored in the local *tocommit_queue* of M^k at the time of early validation are those transactions which have been validated but not committed yet (i.e., category (ii)). This means that $T_j.tid > T_i.start$ for each transaction T_j in the *tocommit_queue* of M^k. Moreover, if at all, T_i will commit after T_j since if T_i is added to the queue it will be appended to the end. Thus, T_i may not conflict with any of these transactions. Therefore, early validation will validate against all transactions in the *tocommit_queue*.

However, early validation cannot include transactions in category (iii), because these transactions have not yet arrived at M^k at the timepoint of early validation. In order to not miss these transactions, we need to perform a second validation of T_i after the delivery of T_i's writeset. This validation has to be performed at all replicas.

Figure 5.3 presents the adjustments of SIMC to handle early validation. We highlight the changes in bold letters. First of all, we do not need the variable $lastcommitted\_tid$ and the mutex $dbmutex$. They were needed to figure out when exactly a transaction started in the database replica. This is no more needed, because we only validate against transactions in categories (ii) and (iii) above. That is, steps 1a and 3(b)iv in SIMC (Figure 5.2) become easier. We do not need to keep track of $lastcommitted\_tid$ at start time. Step 3(b)iv in Figure 5.2 is rewritten to step 3(b)iv and 3(b)v in Figure 5.3.

After step 1(d)iii, that is, after retrieving the writeset, we perform the early validation (step 1(d)iii to 1(d)vi). The transaction is aborted immediately, if the transaction conflicts with one of the transactions in the *tocommit_queue*. Otherwise, the transaction will keep track of the *tid* of the last transaction it was validated against (step 1(d)v) with the variable *vid* and then is multicast. Upon delivery, the transaction is only validated against transactions with $tid > vid$ (step 2b). The rest of the protocol remains the same.

Correctness: We only want to outline that the changes in comparison to the original SIMC do not change the correctness of the system.

The new protocol in Figure 5.3 does not contain $lastcommitted\_tid$ and $dbmutex$ compared to the original SIMC in Figure 5.2. But it still validates an update transaction T_i against the same concurrent transactions as the original SIMC. We have carefully discussed above that all concurrent transactions of T_i will be checked either (i) during the execution of T_i within its local database

replica, or (ii) during early validation in Step 1(d).iii-vi or (iii) at validation after delivery at Step 2. Hence, the new SIMC and the original SIMC reach the same decision in their validation. Hence, the new SIMC should also provide 1-copy-SI+IC.

5.2.6 Fault-tolerance

The decentralized architecture does not have a single point of failure. As we have mentioned in Section 2.3, replication protocols based on group communication can take advantage of the delivery guarantees these systems provide. If a group communication system offers uniform reliable delivery, then a replication protocol can be assured that any message delivered to any replica will also be received by the available replicas. Replication protocols such as [61, 67, 90, 6] take advantage of this to achieve fault-tolerance for the replicated system. That is, they guarantee that whenever a transaction is committed at one replica, it will be committed at any available replica (while crashed replicas have to do so upon recovery).

However, in regard to clients, few approaches indicate how a client handles the failure of the replica it is connected to. [73] describes how failures can be made nearly completely transparent to clients in a protocol such as the SIMC protocol. We briefly repeat the idea here.

We assume clients are connected via a standard interface, such as the JDBC interface, to the middleware. A driver is installed at the client. A driver is a software package that provides to the client the interface, and handles the communication with the server. For fault-tolerance purposes, the driver software needs to know the set of middleware replicas. This can be implemented via a directory service or similar. At start time, the driver connects to one of the middleware replicas but is aware of the other middleware replicas in the system. If there are any changes in the configuration, the middleware replica can inform the drivers that are connected. We assume that the middleware replica and co-located database replica fail as one unit. When a middleware replica crashes all its client connections are lost. The drivers on the clients will detect this and automatically connect to another replica. At the time of the crash the connection might have been in one of the following states.

Initialization:
$next\_tid$:=1, ws_list:={}
$tocommit\_queue$:={}, $wsmutex$
$timeout\_threshold$:=$certain\_value$ (e.g.,50ms)

1. Upon receiving an operation Op_i of T_i
   (a) **if Op_i is start, then**
      i. **begin T_i at R^k**
      ii. **return to client**
   (b) else if Op_i is read or write
      i. execute in local R^k and return to client
   (c) else if Op_i is abort, then
      i. abort T_i at R^k and return to client
   (d) else (commit),
      i. $T_i.WS$:= $getwriteset(T_i^k)$ from local R^k
      ii. if $T_i.WS = \emptyset$, then
         • commit and return
      iii. **obtain $wsmutex$**
      iv. **if $\exists T_j \in tocommit\_queue \wedge T_i.WS \cap T_j.WS \neq \emptyset$**
         • **release $wsmutex$**
         • **abort T_i at R^k and return to client**
      v. **$T_i.vid$:=$next\_tid$-1**
      vi. **release $wsmutex$**
      vii. multicast the writeset of T_i in total order

2. Upon delivering T_i in total order,
   (a) obtain $wsmutex$
   (b) **if $\nexists T_j \in ws\_list$ such that $T_i.vid < T_j.tid \wedge T_i.WS \cap T_j.WS \neq \emptyset$**
      i. $T_i.tid$:=$next\_tid$++
      ii. append T_i to ws_list
      iii. $T_i.timeout$:= $current\_time$
      iv. append T_i to $tocommit\_queue$
      v. release $wsmutex$
   (c) else
      i. release $wsmutex$
      ii. if T_i is local, then abort T_i at R^k and return to client

3. Upon T_i is the first in $tocommit\_queue$,
   (a) if T_i local at R^k,
      i. obtain $T_i.timeout\_mutex$
      ii. $T_i.timeout := 0$
      iii. release $T_i.timeout\_mutex$
   (b) if T_i remote at R^k or $T_i.aborted$=true, loop
      i. begin T_i^k at R^k
      ii. apply $T_i.WS$ to R^k
      iii. if T_i aborted by deadlock, then
         • continue the loop
      iv. **else if T_i aborted by IC, then**
         • **exit the loop**
      v. **else,**
         • **commit T_i at R^k**
         • **exit the loop**
   (c) return to client if T_i is local.
   (d) remove T_i from $tocommit\_queue$

4. Upon timeout of transaction T_i local at R^k, (i.e., $current\_time$ - $T_i.timeout > timeout\_threshold$)
   (a) obtain $T_i.timeout\_mutex$
   (b) if $T_i.timeout \neq 0$, then
      i. abort T_i
      ii. $T_i.aborted$:=true
   (c) release $T_i.timeout\_mutex$

Figure 5.3: SIMC with the early validation optimization

1. There was currently no transaction active on the connection. In this case, failover is completely transparent.

2. A transaction T was active and the client has not yet submitted the commit request. In this case, T was still local on the middleware/DB replica that crashed, and the other replicas do not know about the existence of T. Hence, it is lost. The JDBC driver returns an appropriate exception to the client program. But the connection is not declared lost, and the client can restart T.

3. A transaction T was active and the client has already submitted the commit request which was forwarded to the middleware replica. In this case, the state at the remaining available replicas might be as follows:

   (a) They have not received T's writeset, and hence, do not know about the existence of T, and T must be considered aborted.

   (b) They have received T's writeset. If validation succeeds, they commit T.

   Note that uniform reliable delivery guarantees that if the local replica received the writeset and committed T before the crash, then all (available) remote replicas receive the writeset and hence, also commit T.

Let's have a closer look at case 3. If clients are directly connected to the database and the database crashes after a commit request but before returning the confirmation, clients do not know whether the transaction aborted or committed. In SIMC, we are able to provide the clients with the outcome. When a new transaction starts at a middleware replica, the replica assigns a unique transaction identifier and returns it to the driver. Furthermore, the identifier is forwarded to the remote middleware replicas together with the writeset. Each replica keeps these identifiers together with the outcome of the transaction. If now a crash occurs during a commit request, the JDBC driver connects to a new replica and inquires about the in-doubt transaction by sending the transaction identifier. If the new replica had not received the writeset, it does not know about the identifier, and hence, informs the driver that the transaction did not commit. The driver returns the same exception to the client as if the commit was not yet submitted at the time of crash. If the new replica has the

identifier, it checks for the outcome and returns the outcome to the driver which forwards it to the client program. In this case, failover was completely transparent.

Note that due to the asynchrony of message exchange it might be possible that the middleware receives the inquiry about a transaction from a driver and only after that it receives the writeset for the transaction. In order to handle this correctly, the replica does not immediately return to the JDBC driver if it does not find the transaction identifier. Instead, it waits until the GCS informs it about the crash of the old replica. According to the properties of the GCS, the new replica can be sure that it either receives the writeset before being informed about the crash or not at all. Hence, it can inform the driver accordingly.

5.3 SEQ: a replication protocol without GCS

[72] shows that in WANs the response of a transaction largely depends on the WAN communication overhead. SIMC does not require any WAN communication for read-only transaction (as long as a client has a replica close by). It requires only one multicast message per update transaction, much better than the many WAN message rounds SRP-IC and other centralized replication approaches have per transaction.

However, although the properties of group communication systems are very powerful, there are some disadvantages and problems when using them. First of all, there exists a whole range of total order algorithms each of them having different message overhead and latency. While message overhead and latency do not play a large role in LANs, considering the performance of the total order multicast is extremely important in a WAN.

Furthermore, uniform reliable delivery increases latencies even further, because it typically requires additional acknowledgment rounds before a message is actually delivered to the application. In a WAN this becomes quickly unacceptable.

Finally, there are actually not many stable, publicly available group communication systems available. Indeed, we are only aware of one publicly available system, Spread [114], that provides total order multicast and uniform reliable delivery. Unfortunately, the particular choice of total order

multicast and its implementation of uniform reliability have a very high latency. Most other available systems, such as Ensemble [41] or JGroups [59], only provide reliable delivery. As mentioned in Section 2.2.8, using reliable delivery instead of uniform reliable delivery it might occur that a site receives a message (and, e.g., commits a transaction), and then fails before anybody else has received the message.

Based on these observations, this section proposes a replication protocol SEQ, that integrates the functionality of GCS into the replication architecture. It chooses those techniques developed for GCS that seem the most promising for replication purposes and merges them with the replica control functionality.

5.3.1 Analysis of multicast algorithms

[36] gives a very detailed analysis of different multicast algorithms guaranteeing total order and/or uniform reliability. Here we analyze three of them. Among them, only one provides uniform reliability by default. The others need extra message rounds for uniformity. Table 5.1 shows how message exchange is done in these protocols.

In principle, all protocols assume that there exists a point-to-point protocol that sends a message reliably to the recipient, that is, as long as there are no crashes the receiver receives the message (implemented, e.g., via TCP/IP). Our performance overhead assumes n processes in the system.

In sequencer-based algorithms, one of the processes has the special role of a sequencer. If a process wants to multicast a message in total order, it sends the message to the sequencer. The sequencer gives the message a sequence number and sends the message on behalf of the original sender to all members of the group. All processes deliver messages in the order of their sequence numbers. There are n messages sent in total for one application message, and the delay from sending the message to delivering it is two message rounds. To achieve uniform reliability, all processes, upon receiving a message, send an acknowledgment back to the sequencer. The sequencer sends then a confirmation to all processes. Only upon receiving the confirmation a process can deliver the message to the application (in order of sequence number). Thus, the number of messages increases to $n+2(n-1)$ and the message rounds increase to 4. [18, 22, 63, 83] follow the sequencer approach. JGroups [59] has a variation on the sequencer approach. A process first fetches a sequence number

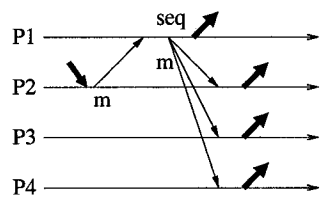
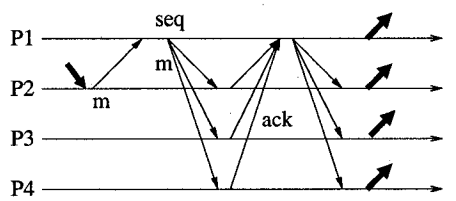
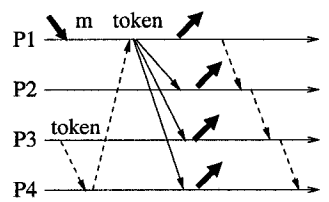
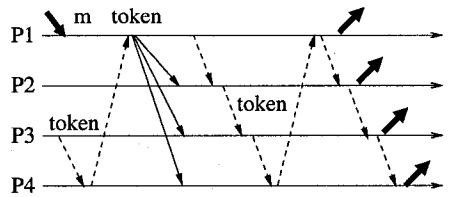
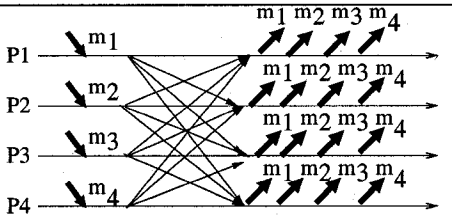
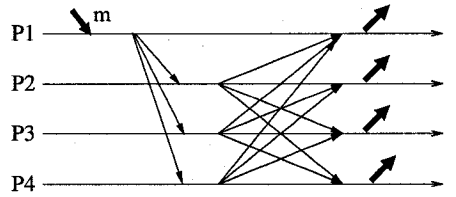
Algorithms	Total order	Total order + uniform reliability
Sequencer-based		
Token-based		
Tmestamp-based		

Table 5.1: Different multicast algorithms (adapted from Table-1 in [36])

for its message from the sequencer and then sends the message with this sequence number to all processes. Thus, the number of messages is $1 + n$ and the number of message rounds is 3 (for reliable multicast).

In token-based algorithms, there is a token circulating among all processes. The token carries the sequence number of the latest message that has been multicast. If a process wants to multicast a message, it waits until the token arrives. Then it takes the sequence number from the token, increases it by one, timestamps the message with this value and sends it to all processes. It does so for all messages it wants to send. Then it adds the sequence number of the last message sent to the token and forwards it to the next process. The number of messages per application message is n (not considering the token messages), and the delay is on average $n/2$ because a node has to wait until it receives the token before it can start sending. To achieve uniform reliability, the token also contains the sequence number of the last message each process has received. This information is used by a process to determine when it is safe to deliver a message, namely when it knows that everybody else has received it. No extra messages are needed to achieve uniformity, but the message delay is increased to $n + n/2$ on average. Spread [114] and Totem [81] are examples of token-based total order algorithms. JGroups [59] provides a total order implementation based on Totem. But it only guarantees reliable delivery instead of uniform reliability.

In timestamp-based algorithms, each message m is timestamped with a vector of n counters showing the number of messages received per process before m is sent to all processes. Each process can order all incoming messages according to their timestamps. It can also determine with the help of these timestamps when other processes have received certain messages. That is, successive messages are implicit acknowledgments for previous messages. This allows a process to deliver a message in total order and when uniform reliability is guaranteed. The number of messages sent per application message is $n - 1$. The number of message rounds to achieve total order and uniform reliability is in the best case 2. Timestamp-based algorithms were proposed in [71] but we are not aware of any group communication system implementing it.

We also consider a total order algorithm which is based on timestamps and does not guarantee uniform reliability. In the algorithm, each process attaches a local sequence number and its process-id to a message before sending it to all processes. Each process delivers messages in round robin

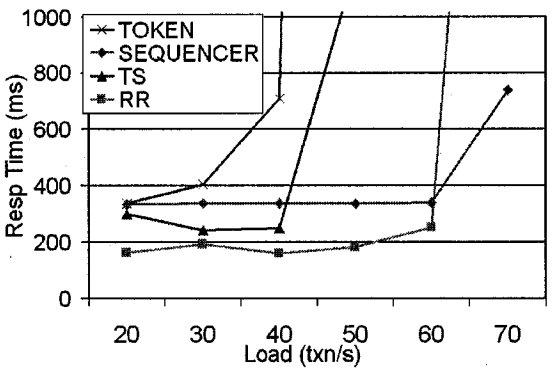


Figure 5.4: Performance of different multicast algorithms with database replication in WAN (Fig. 7 in [72])

mode, i.e., one message from each process in turn. This requires $n-1$ messages per application message, and ideally, if all processes continuously send messages, only 1 message round. For uniform reliability a similar scheme as for sequencer-based algorithms can be used, increasing the message number to $3(n-1)$ and the message rounds to 3.

To see how these algorithms differ from each other when working with database replication, we evaluated them in a WAN with 5 sites (in Montreal, Edmonton, Waterloo, Madrid, Zurich) on top of a protocol similar to SIMC. The nodes had different but similar setup (similar to Pentium(R)-4 CPU 1700MHz, 512MB memory). We only considered reliable, but not uniform reliable delivery since only one available group communication system (Spread) actually provides uniform reliability.

Figure 5.4 shows the average response time of transactions with increasing load submitted to the system. We used the sequencer-based algorithm (SEQUENCER) and token-based algorithm (TOKEN) implemented in JGroups [59]. We provided our own implementations for round-robin (RR) and timestamp-based total order multicast (TS) on top of JGroups. Note that SEQUENCER and TOKEN provided by JGroups only provide reliable delivery guarantee, so does RR. TS provides uniform reliable delivery.

The figure shows that TOKEN has the worst response time due to the circulation of the token. SEQUENCER offers better performance although it requires three messages per application message, and has the potential bottleneck of the sequencer site. Furthermore, it leads to stable response

times until the sequencer becomes saturated at around 70 *transactions per second (tps)*. TS provides faster response times than TOKEN and SEQUENCER for low loads up to 40 tps although this protocol provides additionally uniform reliable delivery. Interestingly, response times at 30 tps are better than at 20 tps because when more messages are sent, the implicit acknowledgments arrive faster. RR has the lowest response time of all since there are no additional messages and message rounds. It saturates only shortly before the sequencer due to CPU overhead. However, RR requires that all processes send messages in regular time intervals. If a process stops sending messages, all other processes will not be able to deliver messages further.

Our analysis shows that the distributed algorithms TS and RR can achieve slightly better performance, however they cannot achieve the same throughput as the sequencer based algorithm. Uniform reliability seems infeasible in a WAN. TS provides uniform reliability but it saturates at very low throughputs. We did not evaluate token-based algoritms with uniform reliablity, since the reliable token-based algorithm (TOKEN) has the worst response time already. Uniform reliable token-based algorithms definitely have much worse response time since they require one more round of token circulation than reliable token-based algorithms.

5.3.2 Basic idea

Our analysis of the previous section shows that uniform reliable delivery seems infeasible in a WAN. However, using only reliable delivery will require the replication tool to be particularly careful in the failure case. Therefore, it makes sense to combine sequencer-based ordering with replica control and develop independent fault-tolerance mechanisms instead of depending on the group communication system.

Recall that SIMC needs total order multicast to guarantee that all writesets are validated in the same order at all replicas so that all replicas make the same decision. We can assign one of the middleware replicas as the unique sequencer in the system. The idea is that instead of multicasting a writeset with total order, a middleware replica sends the writeset only to the sequencer middleware. Only the sequencer middleware performs the validation. If validation succeeds it forwards the writeset to all middleware replicas in FIFO order. If not, it simply sends the abort decision back to the originator. The other middleware replicas now apply the writesets and commit transactions

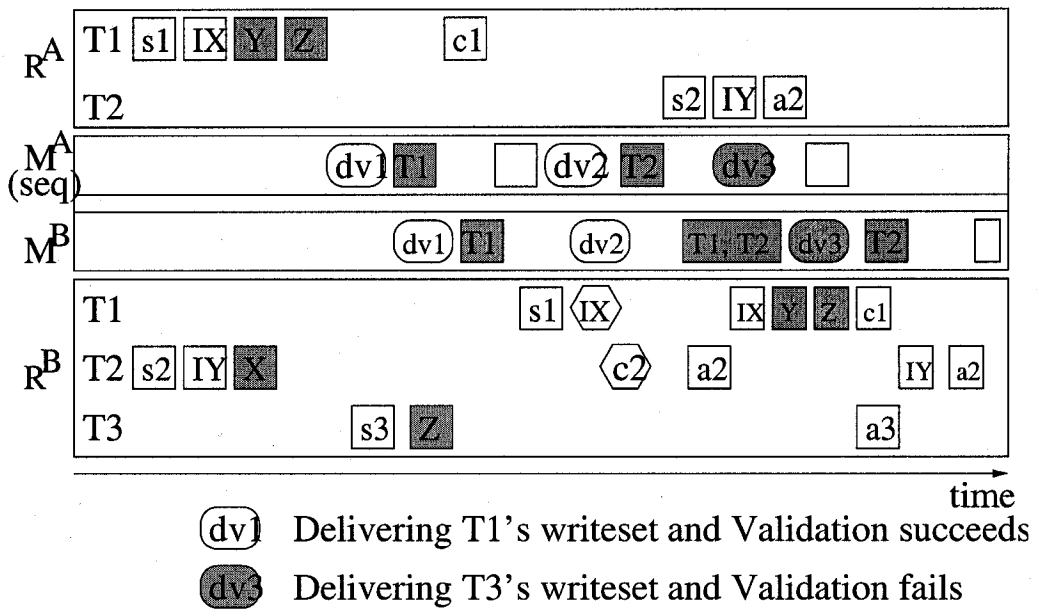


Figure 5.5: Revisit Example 15 in Figure 5.1 using SEQ

in the order they receive them from the sequencer. Integrity constraints and distributed deadlocks are handled just in the same way as in SRP-IC and SIMC. The early validation optimization is still applicable. We refer to the new protocol as SEQ.

5.3.3 Example

Example 16. *Figure 5.5 revisits the example for SIMC (i.e., Example 15). We mainly focus on how validation is performed in SEQ for write transactions.*

We let middleware M^A be the sequencer. After T_1's execution at R^A, T_1 is validated immediately locally at M^A since it is the sequencer. T_1's validation is successful and T_1 is appended to tocommit_queue of M^A. T_1's writeset and validation decision are also sent in FIFO order to M^B. Before its delivery, T_2 and T_3 are executed at R^B. Each validation does not checked any conflict and they are sent to M^B for validation. Then M^B receives T_1's writeset. It does not need to validate and immediately appends T_1 to its tocommit_queue.

In the meantime, T_2 is executed locally at R^B. After its execution, its writeset is sent to the sequencer M^A for validation. At M^A, when M^A receives T_2, it is successfully validated since it

does not have write/write conflicts with T_1. The decision is sent back to M^B. By now both M^A and M^B know that T_2 has been validated successfully. We use the same technique as in SIMC to apply T_1 and T_2 after their validation. T_1 will commit while T_2 will abort due to integrity constraints.

When M^A receives T_3, the validation fails due to the successful validation of T_1. M^A discards T_3 and sends the abort decision to M^B. M^B aborts T_3 at R^B.

5.3.4 Protocol details

Figure 5.6 shows the details of SEQ which is based on the optimized SIMC in Figure 5.3. It highlights the difference with bold letters.

In the local execution phase (step 1), SEQ is the same as SIMC except the last step. A replica sends the writeset of an update transaction to the unique sequencer instead of multicasting it in total order.

Validation can only happen at the sequencer site (step 2). If validation fails, the sequencer only sends the *abort* decision back to the sender (step 2c). Otherwise, the sequencer multicasts the *commit* decision and the writeset to all replicas (step 2b).

Upon receiving a *commit* decision and the corresponding writeset (step 3), a replica appends the transaction to its *tocommit_queue* for execution (step 3b). Upon receiving an *abort* decision (step 4), a replica aborts the corresponding transaction (step 4a). Note that the replica does not need to perform the validation again.

A transaction in *tocommit_queue* will be applied according to the same rules as in SIMC (step 5). The timeout mechanism is also the same as in SIMC (step 6). We do not repeat them here.

Correctness: Since SEQ has a unique sequencer to make the decision to commit or abort update transactions, its proof is similar to that of SRP-IC and omitted.

5.3.5 Fault-tolerance

Fault-tolerance needs a detailed analysis because there is no group communication system and no uniform reliable delivery.

Initialization: $next\_tid$:=1, ws_list:={}
$tocommit\_queue$:={}, $wsmutex$
$timeout\_threshold$:=$certain\_value$ (e.g.,50ms)

1. Upon receiving an operation Op_i of T_i
   (a) if Op_i is start, then
      i. begin T_i at R^k
      ii. return to client
   (b) else if Op_i is read or write
      i. execute in local R^k and return to client
   (c) else if Op_i is abort, then
      i. abort T_i at R^k and return to client
   (d) else (commit),
      i. $T_i.WS := getwriteset(T_i^k)$ from local R^k
      ii. if $T_i.WS = \emptyset$, then
         • commit and return
      iii. obtain $wsmutex$
      iv. if $\exists T_j \in tocommit\_queue \wedge T_i.WS \cap T_j.WS \neq \emptyset$
         • release $wsmutex$
         • abort T_i at R^k and return to client
      v. $T_i.vid$:=$next\_tid$-1
      vi. release $wsmutex$
      vii. **send T_i to M^{SEQ}**

2. **Upon receiving T from M^l (M^{SEQ} only)**
   (a) **obtain $wsmutex$**
   (b) **if $\nexists T_j \in ws\_list$ such that $T_i.vid < T_j.tid \wedge T_i.WS \cap T_j.WS \neq \emptyset$:**
      i. **$T_i.tid := next\_tid$++**
      ii. **append T_i to ws_list**
      iii. **send (COMMIT, T_i) to all middleware replicas in FIFO order**
      iv. **release $wsmutex$**
   (c) **else**
      i. **release $wsmutex$**
      ii. **send (ABORT, T_i) back to M^l**

3. **Upon receiving (COMMIT, T_i) from M^{SEQ}**
   (a) **$T_i.timeout := current\_time$**
   (b) **append T_i to $tocommit\_queue$**

4. **Upon receiving (ABORT, T_i) from M^{SEQ}**
   (a) **abort T_i and return to the client**

5. Upon T_i is the first in $tocommit\_queue$,
   (a) if T_i local at R^k,
      i. obtain $T_i.timeout\_mutex$
      ii. $T_i.timeout := 0$
      iii. release $T_i.timeout\_mutex$
   (b) if T_i remote at R^k or $T_i.aborted$=true, loop
      i. begin T_i^k at R^k
      ii. apply $T_i.WS$ to R^k
      iii. if T_i aborted by deadlock, then
         • continue the loop
      iv. else if T_i aborted by IC, then
         • exit the loop
      v. else (T_i successfully applied)
         • commit T_i at R^k
         • exit the loop
   (c) return to client if T_i is local.
   (d) remove T_i from $tocommit\_queue$

6. Upon timeout of transaction T_i local at R^k, (i.e., $current\_time$ - $T_i.timeout$ > $timeout\_threshold$)
   (a) obtain $T_i.timeout\_mutex$
   (b) if $T_i.timeout \neq 0$, then
      i. abort T_i
      ii. $T_i.aborted$:=true
   (c) release $T_i.timeout\_mutex$

Figure 5.6: SEQ at middleware replica M^k

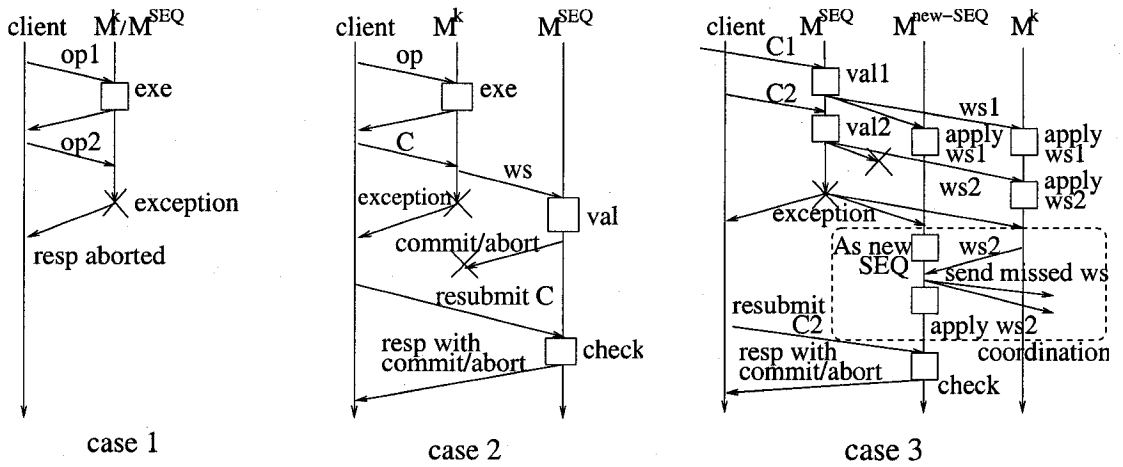


Figure 5.7: SEQ failover cases (I)

The client failover is as in SIMC. The JDBC driver of the client automatically reconnects to a new middleware replica if it looses the connection to its old replica. If there was no transaction active on a connection, nothing special has to be done.

Note that the execution of T might be affected by crashes of its local replica M^k and the sequencer replica M^{SEQ} (if M^k is not M^{SEQ}). The crash of any other replica has no impact on T. We will analyse the failover according to the crashes of different replicas.

Crash of M^k (sequencer/non-sequencer) during execution of T (Figure 5.7 *case 1*)

If T's local replica M^k crashes in the middle of execution of T (i.e., before the client submitted the commit request), then the driver simply returns an abort exception to the client program before reconnecting to a different replica. This is necessary, because the sequencer does not yet know about the transaction, and hence it cannot be recovered. Figure 5.7 *case 1* shows such a scenario. Note that an abort exception will be thrown no matter if M^k is M^{SEQ} or not. In case M^{SEQ} crashes, there is a coordination which will be described later.

Crash of non-sequencer M^k after submitting T's commit request (Figure 5.7 case 2):

Figure 5.7 case 2 shows such a scenario. In this case, the JDBC driver receives a failure exception as return to the commit request. The driver resubmits the same commit request to the sequencer replica. Upon receiving such resubmission, the sequencer checks whether it had received the writeset of the

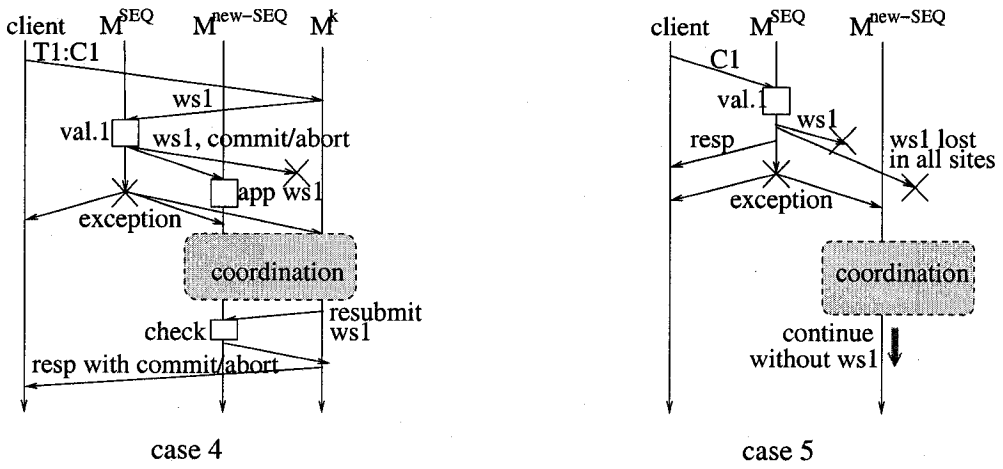


Figure 5.8: SEQ failover cases (II)

corresponding transaction from M^k. If yes, it will return the outcome (commit or abort) to the driver/client. If not, it returns an abort decision because the transaction is lost. From there, the driver can decide to stay connected with the sequencer, or connect to a replica that is closer to the client. In the latter case, the driver has to be careful that it only sends the next transaction to this replica once it can be sure that all previous transactions transmitted through this driver have been applied at this replica in order to guarantee session consistency (a transaction sees the changes of committed transactions from the same client).

Crash of sequencer M^{SEQ}

Corrdination of new sequencer (Figure 5.7 case 3)

First, when the sequencer M^{SEQ} crashes, we assume there is an election protocol that determines a new sequencer $M^{new-SEQ}$. For that, SEQ can use, e.g., the membership features of GCS. That is, all middleware replicas build a GCS group and if a member fails, the GCS automatically informs the others about the crash. GCS's unfirm reliable multicast primitives could be used to decide on the next sequencer. We believe using the GCS for this limited purpose is acceptable considering the properties it provides and the fact that failures occur seldomly. If the membership changes again while failover is still ongoing, the failover procedure is simply restarted.

Figure 5.7 *case 3* shows a detailed example of how the new sequencer coordinates the surviving replicas. M^{SEQ} validates T_1 and T_2 successfully, sends both decisions (including writesets)

to M^k but crashes before sending T_2's decision to $M^{new-SEQ}$. Recall that all replicas execute and commit the same transactions in their *tocommit_queue*. Each replica can inform $M^{new-SEQ}$ about their value of *lastcommitted_tid* once *tocommit_queue* is empty. Let $tid1$ be the value of *lastcommitted_tid* of $M^{new-SEQ}$ and $tid2$ the largest value of any *lastcommitted_tid* received from the other replicas and let M^k be the replica that sent this value. If $tid2 > tid1$, then $M^{new-SEQ}$ is missing some transactions that were received by other replicas. $M^{new-SEQ}$ contacts M^k to retrieve the missing transactions (e.g., T_2 in this case)[1]. $M^{new-SEQ}$ applies these transactions locally and sends them to replicas that miss them. Which transactions to send can be easily determined by the corresponding *lastcommitted_tid* values. From there, normal processing resumes on $M^{new-SEQ}$. The drivers that were connected to the old sequencer and had outstanding commit requests connect to the new sequencer and resubmit the commits.

M^{SEQ} crashes and M^k is not M^{SEQ} (Fig. 5.8 case 4):

Let's consider a case that a client submits its transactions to a non-sequencer replica M^k at the time M^{SEQ} crashes. M^k may have sent a writeset of one of its clients for validation to M^{SEQ} but no replica received the commit/abort decision before the crash, as shown in Figure 5.8 *case 4*. Thus, once a replica M^k has received all the missing transactions from $M^{new-SEQ}$, M^k resends the writesets of outstanding transactions to the new sequencer for revalidation. This is transparent to the client and the JDBC driver. It might happen that M^k is not M^{SEQ} and they crash at the same time. Then the coordination selects a $M^{new-SEQ}$ and the client will be redirected to $M^{new-SEQ}$.

M^{SEQ} crashes and M^k is M^{SEQ} (Fig. 5.8 case 5):

Let's consider the other case that a client submits its transactions to M^{SEQ} when M^{SEQ} crashes. There are two situations where inconsistencies can occur because we do not have uniform reliable multicast. First, as shown in Figure 5.8 *case 5*, a transaction T_1 local to the old sequencer might have committed but nobody received the decision before the crash. The client might have received the commit confirmation. Either we block execution until the old sequencer recovers (i.e., no sequencer takeover) or the transaction is lost since the other replicas continue execution without this

[1]Note that this requires replicas to keep decisions and writesets of committed transactions. Hence, some garbage collection process must be in place to eventually delete writesets once it is assured that all replicas have received them.

transaction.

The second inconsistency might arise when a replica sends a writeset for transaction T for revalidation to the new sequencer, and the new sequencer decides on one outcome (either commit or abort) while the old sequencer had decided on a different outcome. From the client perspective, there is no problem because it never received the first decision of the old sequencer (it was not connected to the old sequencer). Hence, this issue is merely a recovery problem. If the old sequencer decided abort and the new sequencer decided commit, then, upon recovery of the old sequencer, one must make sure that the updates performed by T are transferred during the recovery process since it eventually committed. If the old sequencer committed T but the new aborted it, then, upon recovery, the old sequencer has to undo the changes.

5.4 Hybrid: a replication protocol taking advantage of network topologies

5.4.1 Basic idea

While optimized on performance, SEQ has the shortcoming that if the sequencer crashes, some transactions might be lost. SIMC avoids this problem since it uses the uniform reliable delivery of GCS. However, as we discussed before, this uniform reliable delivery is too costly in a WAN.

However, we can still take advantage of GCS in some configurations. In many applications there exist different sets of replicas, each set being connected via a LAN, while the different sets are separated through a WAN. For example, a Chinese news website might have many replicas in the company's headquarter located in Beijing, a large set of replicas in Shanghai, and then smaller sets of replicas dispersed around the world. For these kinds of applications, we propose the HYBRID approach, which addresses both fault-tolerance and performance issues. An example of its architecture is depicted in Figure 5.9. We assume the replicas can be split into different subsets, each of them being located on a different LAN. We assign one LAN with at least two replicas to be the *primary LAN* and the others as *secondary LANs*.

Within the primary LAN, we use SIMC based on GCS. Since communication is fast in a LAN,

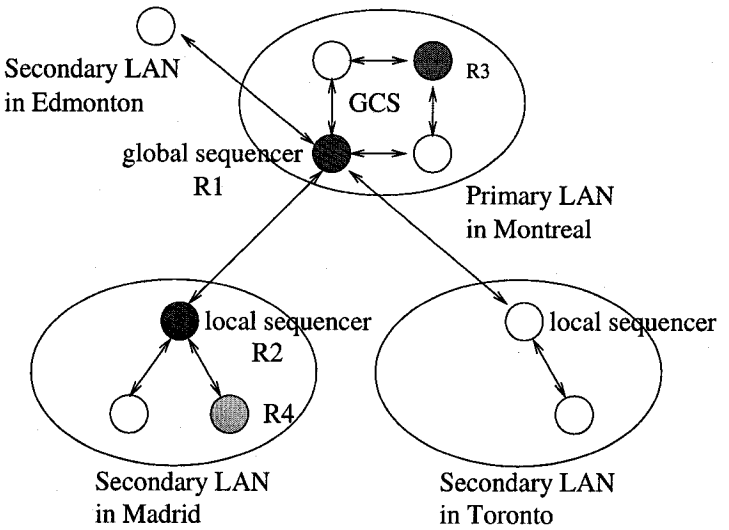


Figure 5.9: An example of network topologies for HYBRID

the overhead of uniform reliable, total order delivery is acceptable. For the secondary LANs, we use hierarchical validation. A transaction is first validated by a local sequencer in the secondary LAN according to the SEQ protocol. Then, if validation succeeds, the local sequencer forwards it to a replica in the primary LAN for further validation. If validation succeeds in the primary LAN, the transaction will be sent to the local sequencers of all secondary LANs which forward it to the other replicas in their LANs. Hence, all replicas apply the writeset. If the global validation fails, the decision is only sent back to the secondary LAN where the transaction originated.

HYBRID improves over SEQ in several ways. First, since the primary LAN uses uniform reliable delivery, no transactions will be lost unless all replicas of the primary LAN crash. Secondly, on the secondary LANs only the local sequencers perform WAN communication, and only these local sequencers must be known in the primary LAN. This also leads to less WAN messages since commit decisions are not sent to all remote replicas but only to the local sequencers which forward them in their local LANs. Moreover, only the sequencer in a LAN will have ports opened on the firewall for WAN access. It reduces the chances for attacks and the complexity of network management. Finally, part of the validation is done at the local sequencers, decreasing the validation load on the primary LAN.

5.4.2 Protocol details

For the sake of simplicity we assume a single replica in the primary LAN to take care of communication with all local sequencers. We refer to this replica as global sequencer (note, however, that validation is done at all replicas in the primary LAN). If this communication overhead becomes too large, the algorithm can be easily extended such that each replica of the primary LAN maintains the communication with some of the local sequencers.

We show the details of the protocol in Figure 5.10. When a transaction is submitted to a replica in a secondary LAN, it follows the same procedure as discussed in SEQ (Step 1) until it passes the validation in the sequencer of the local LAN (Step 2(a)ii). At this time, it can not commit yet because there may be some concurrent conflicting transactions in other LANs. Hence, its writeset has to be sent to the global sequencer in the primary LAN for *global validation.* However, its *vid* value is adjusted so that it will not be validated against those transactions against which it has been validated by the local sequencer. When a transaction is submitted to a replica in the primary LAN (Step 3a), it follows the same procedure as in SIMC. When the global sequencer receives a transaction from a secondary LAN (Step 3b), it multicasts the writeset in uniform reliable and total order within the primary LAN. Thus, all writesets (both from the primary LAN and the secondary LANs) are delivered to all replicas in the primary LAN (Step 3c). They validate transactions according to the delivery order. Thus, all decide on the outcome. If a transaction succeeds in its validation it is enqueued for execution. Moreover, the global sequencer sends in FIFO order the commit decision and the writeset to all the local sequencers of secondary LANs (step 3(c)iD) which forward them to the others replicas of their LANs (Step 2b). Thus all replicas will execute and commit the transaction. If validation fails (Step 3(c)ii) and it was a transaction of the primary LAN, the corresponding replica aborts the transation. Otherwise, the global sequencer notifies the local sequencer of the originator of the transaction about the abort (step 3(c)iiB). This local sequencer forwards this decision to the originator (Step 2c). Replicas on the primary LAN apply writesets as in SIMC (Step 3d).

Initialization: same as SEQ

1. If M^k on secondary LAN:
   - Same as SEQ in Figure 5.6
2. If M^k=$M^{localSEQ}$ on secondary LANs (besides step 1):
   (a) Upon receiving T_i from M^l in the same LAN
      i. obtain $wsmutex$
      ii. if $\nexists T_j \in ws\_list$ such that $T_i.vid < T_j.tid \wedge T_i.WS \cap T_j.WS \neq \emptyset$:
         - $T_i.vid := next\_tid$-1
         - release $wsmutex$
         - send T_i to $M^{globalSEQ}$
      iii. else
         - release $wsmutex$
         - send $(ABORT, T_i)$ back to M^l

   (b) Upon receiving (COMMIT, T_i) from $M^{globalSEQ}$
      i. obtain $wsmutex$
      ii. $next\_tid := T_i.tid$+1
      iii. append T_i to ws_list
      iv. send (COMMIT, T_i) to all M^l in the same LAN in FIFO order.
      v. release $wsmutex$

   (c) Upon receiving (ABORT, T_i) from $M^{globalSEQ}$
      i. send (ABORT, T_i) to the originator of T_i
3. If M^k on the primary LAN:
   (a) Upon receiving an operation Op of T_i
      - Same as SIMC with optimizqation Figure 5.3
   (b) Upon receiving T_i sent by a $M^{localSEQ}$ from a secondary LAN ($M^{globalSEQ}$ only)
      i. Multicast T_i in primary LAN in uniform reliable and total order.
   (c) Upon receiving T_i multicast in the primary LAN in uniform reliable and total order
      i. if $\nexists T_j \in ws\_list$ such that $T_i.vid < T_j.tid \wedge T.WS \cap T_j.WS \neq \emptyset$:
         A. $T_i.tid := next\_tid$++
         B. append T_i to ws_list
         C. append T_i to $tocommit\_queue$
         D. if M^k is $M^{globalSEQ}$, send (COMMIT,T_i) to all $M^{localSEQ}$ in FIFO order
      ii. else
         A. if T_i local, abort T_i and return
         B. else if T_i originated on secondary LAN and M^k is $M^{globalSEQ}$, send (ABORT, T_i) back to the $M^{localSEQ}$ of the originator of T_i.
   (d) Upon T_i is first in $tocommit\_queue$
      - Same as SIMC with optimization (Figure 5.3)
   (e) Upon a local transaction T_i timeout after its successful validation,
      - Same as SIMC with optimization (Figure 5.3)

Figure 5.10: HYBRID protocol on middleware replica M^k

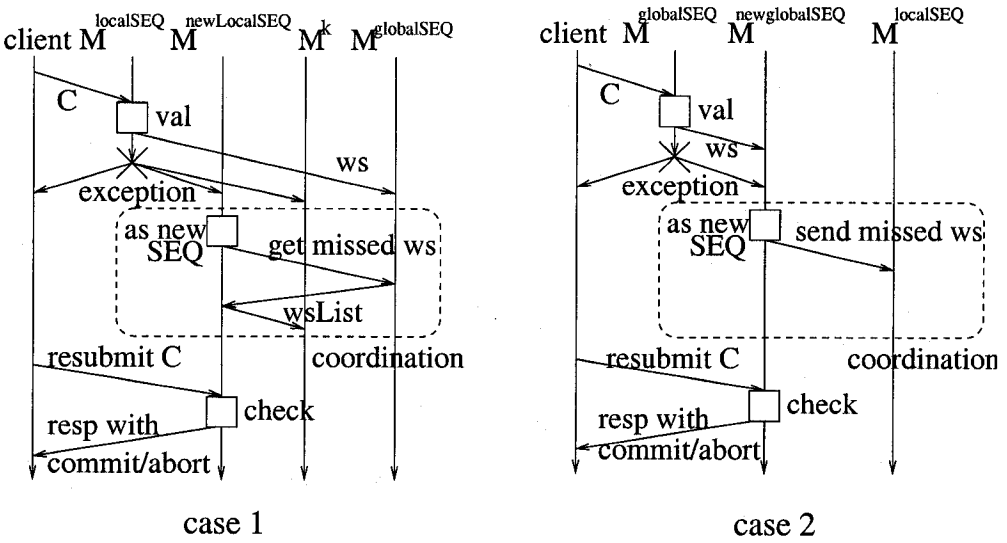


Figure 5.11: HYBRID failover cases

5.4.3 Fault-tolerance

Similarly to SEQ, HYBRID uses a fault-tolerant driver to handle failover. It is an extension of what had to be done for SEQ (Section 5.3.5). When a site crashes before a transaction submits its commit, the driver simply informs the client about an abort exception as shown in Figure 5.7 *case 1* before reconnecting to another replica. The more interesting case is when the client had already submitted the commit request for a transaction but not yet received a response when its local site crashes.

Crash of a non-sequencer replica in secondary LAN: It is the same as described for a non-sequencer replica in the SEQ algorithm (see Figure 5.7 *case 2*).

Crash of a non-sequencer replica in the primary LAN: It is similar to the actions described for a non-sequencer replica in the SEQ algorithm (see Figure 5.7 *case 2*). The driver can reconnect to any replica in the primary LAN. Uniform reliable multicast guarantees that either all or none of the available replicas have received the transaction's writeset, and hence, all make the same decision. It is similar to the discussion in Section 5.2.6.

Crash of the local sequencer $M^{localSEQ}$ in a secondary LAN: An example of this situation is shown in Figure 5.11 *case 1*. As for SEQ, all replicas in the secondary LAN first perform some coordination to decide on a new local sequencer$M^{newLocalSEQ}$. All non-sequencers now inform

$M^{newLocalSEQ}$ about the last writesets they received. Different to SEQ, $M^{newLocalSEQ}$ now informs the global sequencer $M^{globalSEQ}$ about the last transaction it has committed (taking the value of $lastcommitted\_tid$). $M^{globalSEQ}$ sends $M^{newLocalSEQ}$ all the transactions the local sequencer has missed. $M^{newLocalSEQ}$ provides each non-sequencer replica M^k in the LAN with the transactions M^k has missed. As in SEQ, a driver connected to the crashed $M^{localSEQ}$ reconnects to $M^{newLocalSEQ}$ and resubmits the commit request if necessary. And a non-sequencer replica M^k resubmits outstanding writesets to $M^{newLocalSEQ}$. Recall that in SEQ, transactions from clients connected to the crashed sequencer might be lost if their writesets were not transmitted to other replicas before the crash. This problem cannot happen here. Before committing locally, a local sequencer sends its own writesets to the global sequencer. That is, if $M^{localSEQ}$ has committed a transaction before the crash, so has $M^{globalSEQ}$ and $M^{newLocalSEQ}$ will receive it from $M^{globalSEQ}$.

Crash of the global sequencer $M^{globalSEQ}$ in the primary LAN: The global sequencer in the primary LAN is simply the connection point for secondary LANs but validation is actually done by all replicas in the primary LAN. Even if the global sequencer has decided on a transaction but not sent the commit/abort decision to the secondary LANs, all other replicas in the primary LAN are guaranteed to have received the writeset and decided the same outcome. Thus, as shown in Figure 5.11 *case 2*, local sequencers can reconnect to the new global sequencer (which can be easily decided on via voting or pre-selection) and retrieve any missing writesets. The client management is similar to the previous case. As long as one replica survives in the primary LAN, the inconsistency problem that can occur in SEQ is avoided.

If a whole secondary LAN crashes, clients can reconnect to the primary LAN without any loss of transactions. Loosing the full primary LAN would be a catastrophic failure. A secondary LAN should take over but some transactions submitted on the primary LAN might be lost.

5.5 Discussion

In this chapter, we presented three protocols (i.e., SIMC, SEQ, and HYBRID) that are able to execute in WANs and provide fault-tolerance.

SRP-IC does not perform well in WANs due to the centralized middleware architecture it uses.

It requires several message rounds within the response time of a transaction. We solve the problem by using a decentralized middleware architecture (Figure 2.1.(c)). The decentralized architecture introduces additional challenges for validation. SIMC overcomes the problem by using total order multicast provided by GCS. Additionally, uniform reliablity multicast provides fault-tolerance for SIMC. SIMC only requires one multicast message per transaction through the WAN. However, GCS is costly in WANs. Hence, we derive SEQ by discarding the usage of GCS in SIMC. There is a sequencer site in SEQ. The middleware replica at the sequencer site is responsible for all the validation. SEQ only requires two message rounds through the WAN.

We also discuss the fault-tolerance issues in SIMC and SEQ. SIMC can take advantage of uniform reliable multicast semantics provided by GCS. Since SEQ does not rely on GCS, it needs its own fault-tolerance tool. Since this does not provide uniform reliable delivery, there might be a case which might lead to lost transactions.

HYBRID is proposed to overcome this problem as optimization on cluster-based WAN configurations. It is a mixture of SIMC and SEQ. It groups replicas into several groups depending on their network distance (e.g., replicas in one LAN can be one group). A group is designated as primary LAN and SIMC is applied. The remaining clusters are considered as secondary LANs and SEQ is applied. Between primary and secondary LANs, an adjusted SEQ is used. HYBRID can take advantage of uniform reliable multicast to improve the fault-tolerance in the primary LAN. At the same time, it does not experience long message delay incurred by uniform reliable and total order multicast across different LANs.

SEQ and HYBRID both count on their centralized components, i.e., sequencer or primary LAN respectively, to make a final decision of validation. In WANs, network partitioning might sometimes happen and the centralized components might temporarily not be accessible. To avoid secondaries to wait forever, we should set a threshold waiting time at secondaries. The threshold value can be adaptive to empirical data. After timeout, secondaries can either stop execution pessimistically or select a new primary to continue if they can get a quorum of all replicas. In the latter case, the old primary should be discarded.

Protocols	architecture	communication	WAN overhead
SIMC	purely decentralized	total order and uniform reliable multicast	one multicast message
SEQ	decentralized with one sequencer	TCP/IP	two message rounds
HYBRID	SIMC in primary LAN, SEQ in secondary LANs, SEQ between primary and secondary sequencers	total order and uniform reliable multicast within primary LAN, TCP/IP within secondary LANs and between primary and secondary LANs	two WAN message rounds and one LAN multicast message

Table 5.2: Comparison of protocols for WANs

The characteristics of SIMC, SEQ, and HYBRID are summarized in Table 5.2.

Chapter 6

Evaluation

This chapter provides a detailed evaluation of the protocols of Chapter 5, namely SIMC, SEQ and HYBRID. We do not consider the protocols SRP, SRP-IC of Chapter 4 since they do not consider fault-tolerance. However, we compare against two variations of a lazy primary copy approach. They represent typical execution scenarios of existing protocols in terms of execution flow between client and middleware, and middleware and underlying database systems, and thus, allow us to compare our protocols against existing ones in terms of performance.

The remainder of the chapter is structured as follows. First, Section 6.1 describes our replication framework into which we plugged the various replication algorithms. Section 6.2 describes the comparison protocols. Section 6.3 presents two benchmark applications that are used in the experiments. Section 6.6.1 discusses the experimental setup. In Section 6.5, the protocols are evaluated in a LAN environment. Section 6.6 evaluates the performance in a WAN. All experiments are conducted in real networks.

6.1 Replication framework

We have built a middleware-based framework, **MiddleSIR** (**Middleware**-based **S**napshot **I**solation **R**eplication), which accomodates the implementation of different replication protocols. The framework follows the decentralized architecture of Figure 2.1.(c). The inner structure of one middleware

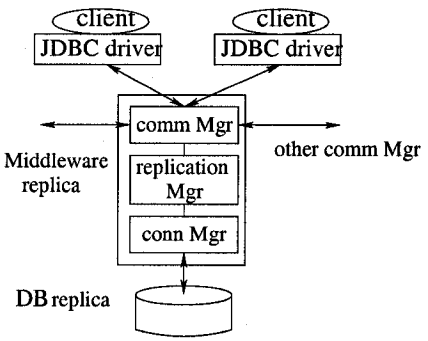


Figure 6.1: MiddleSIR framework

replica is shown in Figure 6.1. A middleware replica is divided into three components, namely, *communcation manager, transaction manager, and connection manager.*

The communication manager is in charge of any kind of communication, including (i) communication between client and middleware, and middleware and database; and (ii) communication between different middleware replicas. Clients submit their requests to the middleware through some standard database interface such as Java Database Connectivity (JDBC). The communication manager interacts with the transaction manager for replica control. The transaction manager takes charge of transaction synchronization. The main part of the replication protocols are implemented in this component. It will detect concurrency and conflicts between transactions and decide whether to commit transactions or abort them. A transaction manager will contact its local connection manager for physically executing, committing or aborting a transaction.

Note that each componenent has different versions of implementation. For example, there are different communication managers according to different communication paradigms, e.g., socket or multicast with group communication. There are different transaction managers corresponding to different replica control algorithms. There are different connection managers corresponding to different underlying DBMSs[1].

In our experiments, we use several kinds of communication paradigms. We use TCP/IP socket communications in SEQ, lazy primary copy approaches, and HYBRID. We use the group communication systems Spread [114] and JGroups [59] in SIMC and HYBRID.

[1]Currently we only implement a connection manager for PostgreSQL

6.2 Comparison lazy primary copy protocols

Our comparison protocols are also implemented in the replication framework, thus relying on one middleware replica for each database replica.

Recall that in a lazy primary copy approach, an update transaction T must be submitted or forwarded to the primary replica for execution which propagates then the changes made by T to the secondary replicas lazily, that is, after committing T. Read-only transactions can be executed at any replica.

We consider two lazy primary copy protocols. LPnMsg is more suitable for a LAN since it has a considerable message overhead. LP1Msg has only two message rounds per transaction between the middleware replicas and thus, is more suitable for a WAN. However, in this case, the middleware does not provide a standard JDBC interface to the application. Instead, the middleware must know all transactions, and receives from the client a request to execute a certain transaction with a specific set of input parameters. Thus, LP1Msg only works if the middleware instance and the application environment are actually collocated. Both protocols require that a transaction indicates at its start time whether it is an update or a read-only transaction.

LPnMsg

When a transaction (read-only or update) is submitted to the middleware replica of the primary replica, the middleware instance simply forwards all requests to the local database replica. When the client submits the commit request, the middleware replica first retrieves the writeset from the database replica, then commits the transaction locally, and finally multicasts the writeset in FIFO order to the secondary replicas.

Secondary replicas simply apply writesets in the order they receive them from the primary replica. For a read-only transaction submitted to a secondary replica, the middleware simply forwards all operations to the local database replica and commits the transaction locally. For an update transaction, the middleware forwards each operation submitted by the client to the primary replica. Note that also read operations have to be forwarded in order for them to read from the proper snapshot. The primary middleware submits it to its local replica and returns the result to the secondary

middleware which forwards it to the client. At commit time, the primary middleware commits the transaction locally and forwards the writeset to all secondaries as it does for writesets of local transactions.

LP1Msg

For transactions submitted to the primary replica, and for read-only transactions submitted to secondary replicas, the protocol works the same as LPnMsg. Also, secondary replicas apply writesets received from the primary sequentially as in LP1Msg.

When an update transaction is submitted to a secondary replica, it is assumed that the entire transaction is submitted by the client in one message. This could be simply a transaction identifier with some parameter values (and the code for the transaction is actually integrated into the middleware itself) or a set of SQL statements. The secondary then forwards the request to the primary middleware which initiates the execution of the transaction at its local database replica, commits the transaction locally, and then forwards the writesets to all secondary replicas where they are applied.

6.3 Benchmarks

6.3.1 TPC-W

TPC-W [118] is a standard benchmark proposed by the Transactional Processing Performance Council (TPC) [119] for E-commerce applications that require a transactional persistent storage. The benchmark simulates an online bookstore. Clients can browse, shop, and order books online. There are three kinds of workloads that vary in the ratio of update vs. read-only transactions (Browsing: 5%, Shopping:20%, Ordering:50%). The TPC-W database consists of 8 tables. The size of each table is determined by the number of items and emulated browsers (clients) in the system. The experiments use a standard setup of 100,000 items and 100 emulated browsers which leads to a database with 650 MByte. The evaluation uses a Java implementation of the benchmark from the University of Wisconsin-Madison [123].

The TPC-W evaluates both web- and database server. Since we are only interested in the behavior of the database, we first generated transaction traces by running the TPC-W using a single web

server, single database server configuration. These traces were then used as input for the evaluation of the replication protocols. In all experiments, the load was evenly distributed to all replicas.

6.3.2 Synthetic benchmark

The second benchmark is a synthetic benchmark. It is used to simulate update intensive workloads (i.e., 100% updates). The evaluation of such a benchmark is useful since replica control is mainly concerned with synchronization of update operations. There are ten tables in the database, each with 10,000 records. Each table has five attributes (two integers, one 50-character string, one float, and one date). The overall tuple size was slightly over 100 bytes, which yielded a database size of just more than 10 MBytes. An update transaction has ten update operations, each of which updates a tuple indexed by a random primary key. Each operation has the form

```
UPDATE table-i SET attr1="randomtext", attr2=attr2+4 WHERE t-id=random(1-10000).
```

6.4 Experimental setup

In each test run, each replica has the same number of clients connected to it. Within a transaction, each client submits the next SQL statement immediately after receiving the previous one, but it sleeps between two different transactions. Each client submits 1000 transactions at the rate of 1 transaction per second in LANs and 0.5 transaction per second in WANs. The number of clients determines the system-wide load. All tests achieved a confidence interval of 95% +- 2.5%. Unless otherwise stated, the timeout value to detect distributed deadlocks was set to 100 ms.

6.5 Local area network

This section analyzes the behavior of the protocols in a LAN. HYBRID is not considered since it is designed for WAN setups in which there are several inter-connected LANs. In our experiments each computer in the cluster has an Intel Pentium-IV CPU with 2.66GHz and 512KB cache, 512 MByte memory, and 30 GB hard disk. Each computer runs the Linux operating system with the kernel of 2.6.17-gentoo-r4. All computers are connected by a 100Mbps Ethernet.

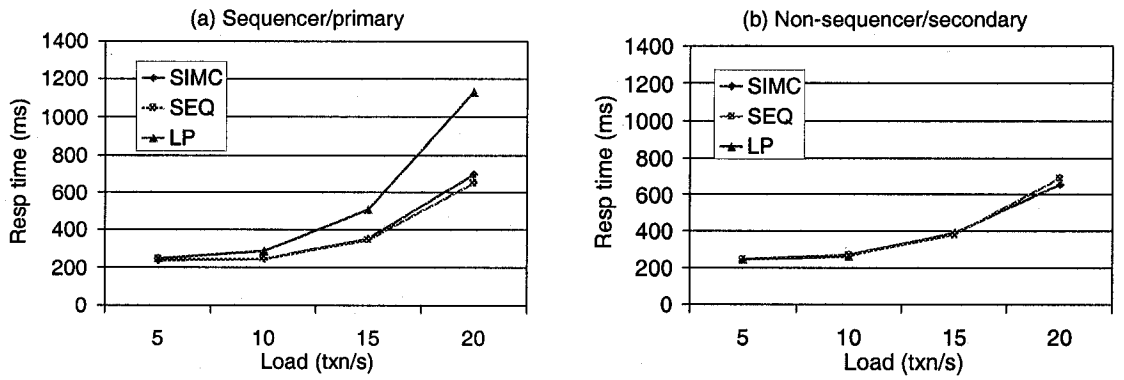


Figure 6.2: Average response time of read-only transactions, TPC-W shopping workload

6.5.1 Base comparison using TPC-W

In this section, we provide a first comparison of SIMC and SEQ with LPnMsg using five replicas. We chose LPnMsg over LP1Msg because it represents the more flexible protocol allowing for a standard JDBC interface. In this section we refer to LPnMsg as LP for simplicity. Recall that SIMC provides purely distributed synchronization, while SEQ and LP both have a node with special tasks (the sequencer in SEQ, and the primary in LP). The workload of the sequencer/primary is different from the other replicas. Thus, the figures separate the results for this special node from the results obtained at the other nodes. Of course, for SIMC, the results are always the same for both node types. SIMC uses Spread providing total order and uniform reliable delivery.

We first use the TPC-W shopping workload with 20% updates. Figures 6.2.(a) and 6.2.(b) show the average response time of read-only transactions at the sequencer/primary replica and the non-sequencer/secondary replicas, respectively, with increasing load. All response times increase with increasing load. At low loads all the protocols behave the same. When the load is increasing, LP is significant worse than SIMC and SEQ at the primary; at the secondaries it is worse only at very high load.

Since a read-only transaction executes only locally and does not trigger any communication, the response time is solely determined by the CPU usage at the local replica. The CPU usage is shown in Figures 6.3.(a) and (b). The figures indicate that SIMC has almost the same CPU usage as SEQ which increases linearly with the load. Correspondingly Figures 6.2.(a) and (b) show the

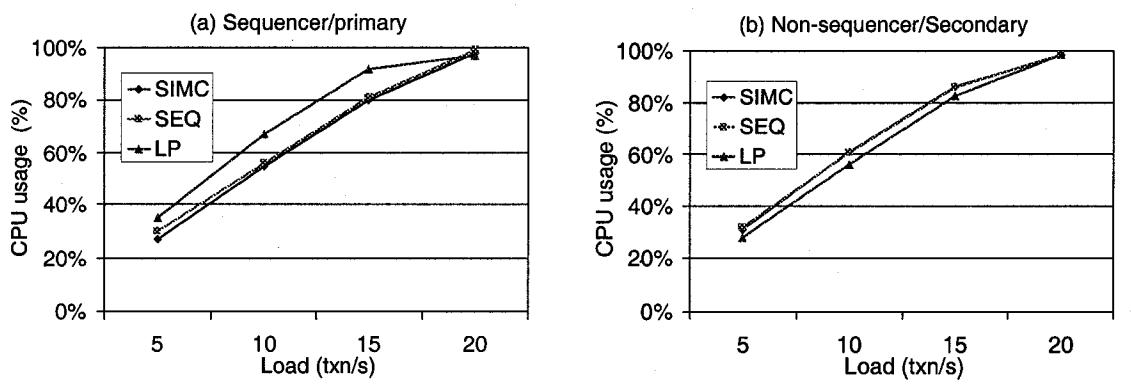


Figure 6.3: CPU usage, TPC-W shopping workload

same average response times for all settings. This means the GCS overhead is similar to the one of TPC/IP socket communication in a LAN, and the asymmetric load of SEQ has no effect on its performance.

Figures 6.3.(a) and (b) also show that LP has higher CPU usage than SIMC and SEQ at the primary replica and slightly lower load at the secondary replicas. This is because SIMC and SEQ have a better load balancing potential than LP. Recall that in LP the primary has to execute all operations (read and write) of all update transactions while secondary replicas only apply the writesets. Furthermore, executing the SQL update statements is more expensive than applying the writeset. In our implementation, applying the writesets at the secondary takes only around 20% of the time it takes to execute the entire transaction at the primary. In contrast, using SIMC or SEQ, update transactions can be executed anywhere, distributing the cost of executing the read and update SQL statements within update transactions across all replicas. Therefore, LP has a much higher load at its primary due to the accumulated load of update transactions, and slightly less load at the secondaries. This leads to observed average response times in Figures 6.2.(a) and (b).

Let's now look at update transactions. Figures 6.4.(a) and (b) show the average response times of update transactions with increasing load for primary/sequencer and secondary/non-sequencer replicas, respectively. In both figures, SIMC and SEQ have low response time up to the saturation point. SIMC has slightly larger response time than SEQ at the sequencer. This is due to the fact that there is no communication delay for the update transactions at the sequencer in SEQ because

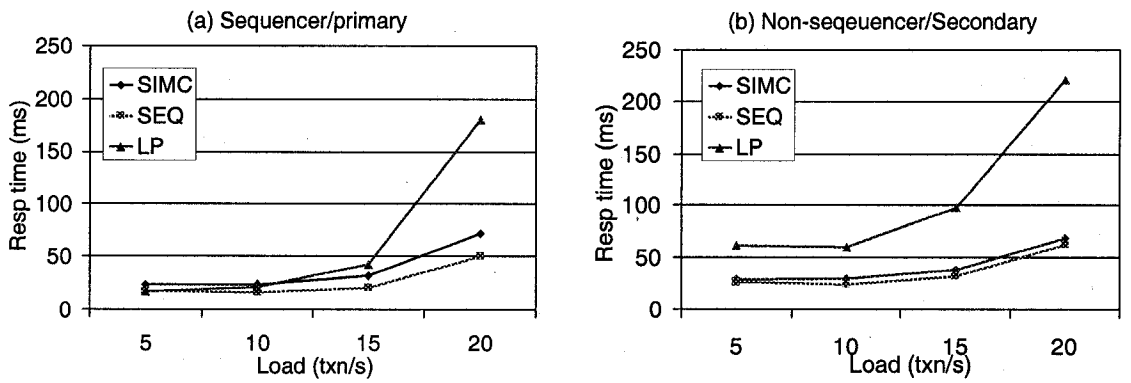


Figure 6.4: Average response time of update transactions, TPC-W shopping workload

they are validated locally. SIMC, however, includes a total order message round. But even at the non-sequencer nodes SIMC is slightly worse although also SEQ includes two message rounds. The reason is that SIMC also provides uniform reliable delivery, not provided by SEQ, that leads to further delay.

At the primary replica, LP's response time is the same as SEQ's response time for a low load but then increases and is significant worse at the saturation point. At low loads LP behaves very similarly to SEQ since the execution pattern is similar and LP is not yet highly loaded. At higher loads LP is simply more loaded leading to worse response times. At the secondaries, LP is significantly worse than the other two protocols. This is because each operation of an update transaction needs to be sent to the primary leading to several message rounds per transaction. A typical TPC-W update transaction has on average four operations, which results in four round trip messages within the response time of the transaction.

We also conducted experiments using the browsing and ordering workloads of the TPC-W benchmark. The results are shown in Figures 6.5 and 6.6. The results show the same tendencies as the shopping workload and thus, will not be discussed in more detail. The behavior of LP compared to the other two is less extreme for the browsing workload since it has mainly read-only transactions, and more extreme for the ordering workload since it has more updates that have to be executed at the LP.

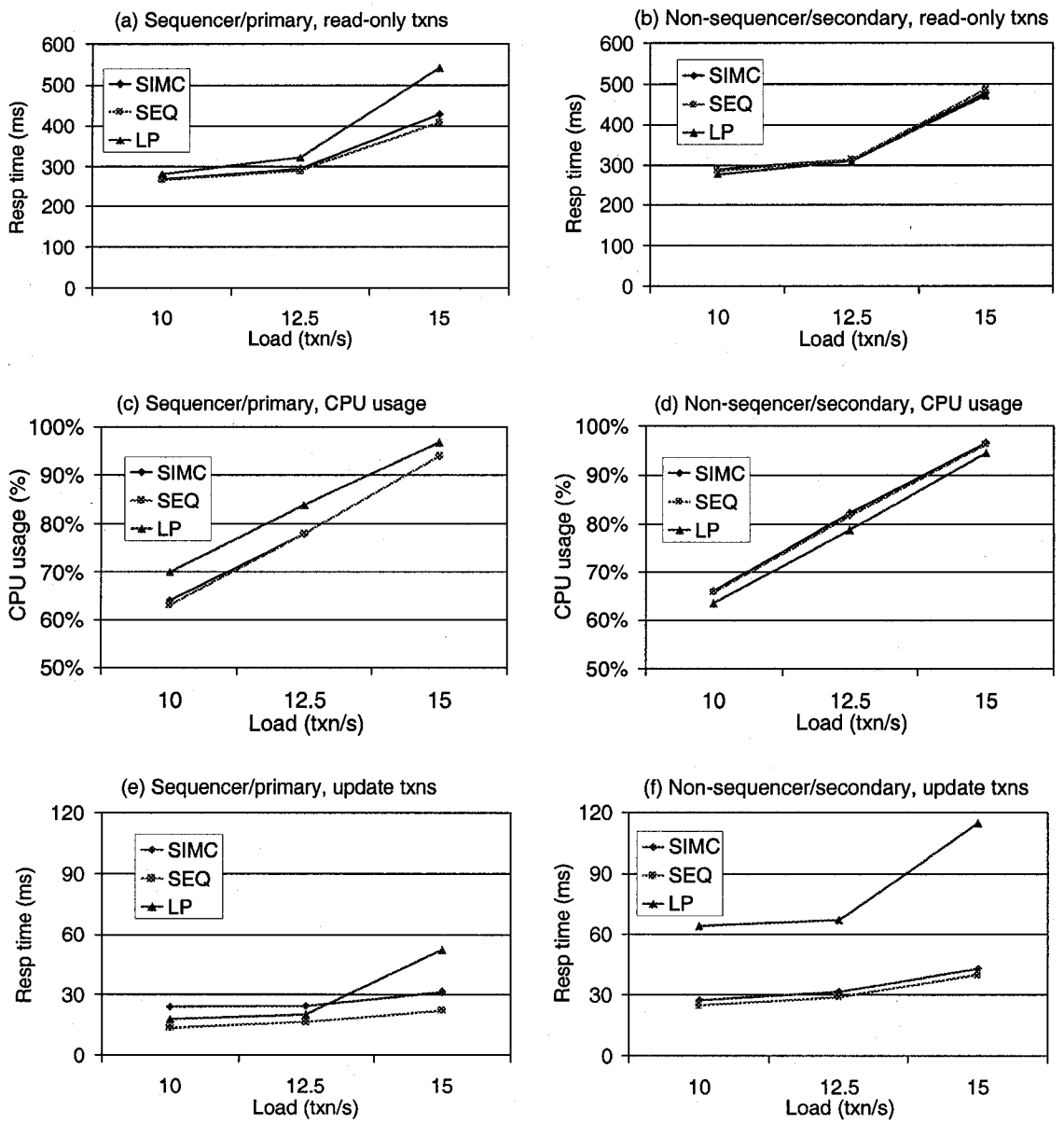


Figure 6.5: TPC-W browsing workload

6.5.2 Stress test using update intensive workload

For all TPC-W workloads the percentage of read operations is fairly high. In this section, we want to stress test the system by using the update intensive synthetic benchmark consisting of 100% updates. This helps to analyze how the replica control component can handle peak situations.

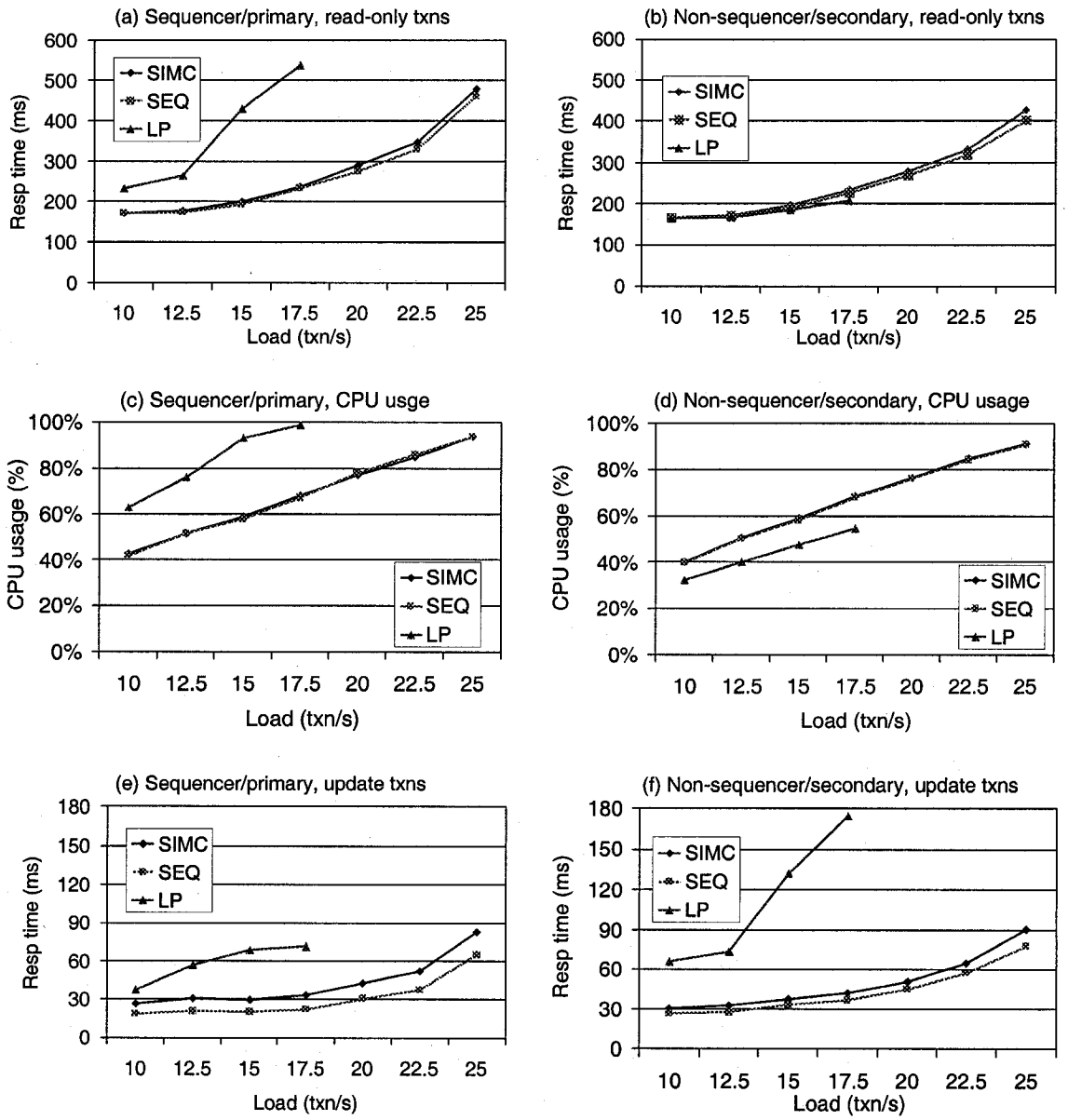


Figure 6.6: TPC-W ordering workload

In this experiment, additionally to SIMC, SEQ and LP with 5 replicas, we also consider a single-node, non-replicated system. Figure 6.7.(a) shows the average response time of the protocols with increasing load. For LP, the figure shows average response times for both the primary and the secondaries. The figure shows that at low load (less than 50 transactions per second) the non-replicated

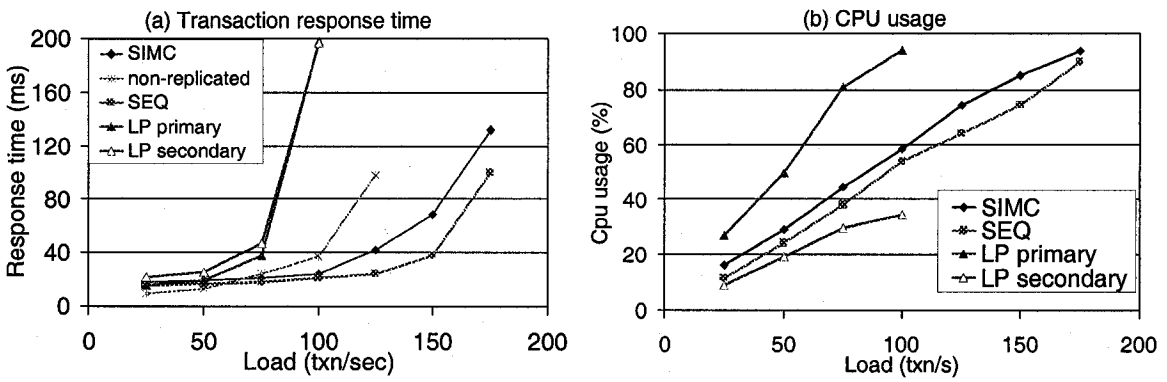


Figure 6.7: Overhead of replication, synthetic benchmark, 100% update

system provides the best performance. This is the expected behavior since a non-replicated system has no CPU or message overhead for replica control. Surprisingly, as the load increases, SIMC and SEQ have lower response times than the non-replicated system. This is surprising since in ROWA all replicas have to execute all updates, and thus, we would not expect any performance gain with replication. However, recall that in Section 6.5.1 we noted that applying the writeset has less cost than executing the SQL update statements. Thus, by having more replicas, instead of each replica executing all SQL update statements, only a subset of them is executed and the rest of updates comes in form of writesets. This observation was already made in [67].

In contrast to SIMC and SEQ, LP is worse than the non-replicated system. Since all transactions are update transactions, the primary is the only one executing transactions. Additionally, it has to forward writesets to the secondaries and handle the operation requests sent by the secondaries to the primary. This raises the overhead well over a non-replicated system and it saturates very fast. The response time at secondaries is even worse than at the primary because they have to send the requests to the primary, let it execute at the overloaded primary and then wait for the response. Thus, additional communication delay is added to the response time. This behavior is confirmed by looking at the CPU overhead in Figure 6.7.(b). The primary replica in LP has the highest CPU load because it has to execute all transactions, and the secondary replicas in LP have the lowest CPU load because they only apply writesets. SIMC and SEQ are in between because each replica executes some transactions and applies the writesets of the others. SIMC has slightly higher CPU load than

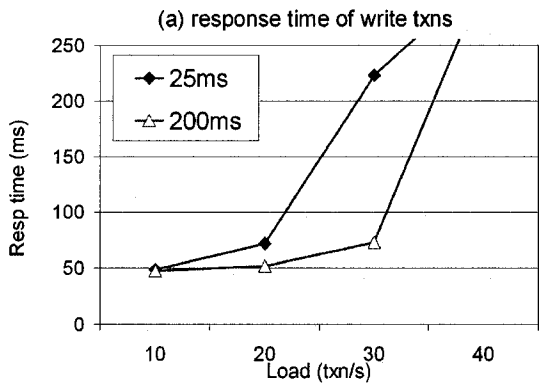


Figure 6.8: Effects of timeout values, shopping workload (I)

SEQ because of the GCS used in SIMC.

The different CPU usage for SIMC and SEQ explains the difference in response time seen in Figure 6.7.(a). At higher loads, SEQ is slightly better than SIMC because its CPU is less loaded.

6.5.3 Effect of timeout values

SIMC and SEQ depend on timeout to detect deadlocks. It is interesting to see how different timeout values affect the performance of the protocol. Recall that the timeout does not span the entire transaction execution but it measures the time interval between the time point a transaction is put into the *tocommit_queue* of a replica until it is the first in the queue.

Figure 6.8 shows the average response time of updates transactions in the TPC-W shopping benchmark with two different timeout values (25 and 200 milliseconds) with increasing load for SIMC. All experiments in this section are carried out at 10 replicas. At low load the performance is the same for both values because basically no transactions wait longer than 25 milliseconds. At higher load, however, the response time for the 25 ms timeout value increases much more sharply than for 200 ms. The reason is that the value is too low letting the middleware assume that there is a deadlock although there is none. This leads to unnecessary aborts and reapplication of the transaction, and thus, longer response times.

Figure 6.9.(b) shows the timeout rate as the percentage of update transactions experiencing timeout with increasing load. Using 200 ms timeout, there are basically no timeouts before the

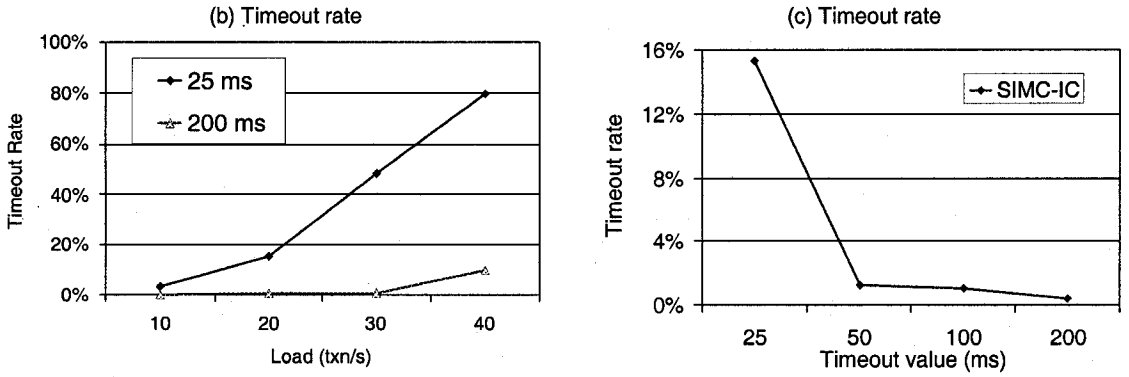


Figure 6.9: Effects of timeout values, shopping workload (II)

saturation point at 40 txn/s. This means, there are, in fact, very few deadlocks in the system. In contrast, with 25 ms there are many timeouts, all of them being false alarms since by choosing a higher timeout value the transactions can actually succeed. Thus, 25 ms is simply a too short timeout interval since it aborts transactions that are not involved in a deadlock. The question is what is the right timeout value so that one has not too many false alarms but one also does not wait too long when actually a deadlock occurs. Figure 6.9.(c) shows the timeout rate with increasing timeout value at a load of 20 txn/s. One can see that the timeout rate drops significantly from 25 ms to 50 ms and then levels off. This shows, that more than 15% of transactions wait in the *tocommit_queue* longer than 25 ms while only around 1% wait for more than 50 ms. 50 ms is around the average response time at this load. Thus, a guideline might be to choose as timeout value according to the average response time for transactions.

6.5.4 Scalability

This section analyzes how SIMC scales in a LAN environment. Figure 6.10 shows the maximum achievable throughput for the three different TPC-W workloads when the number of nodes increases from one to 40. The throughput is generally the highest for ordering, slightly lower for shopping, and the lowest for browsing. The reason is that the read-only transactions in the TPC-W benchmark are more complex and require more resources than the update transactions. Therefore, the more read intensive the workload is, the less transactions can be executed per time unit.

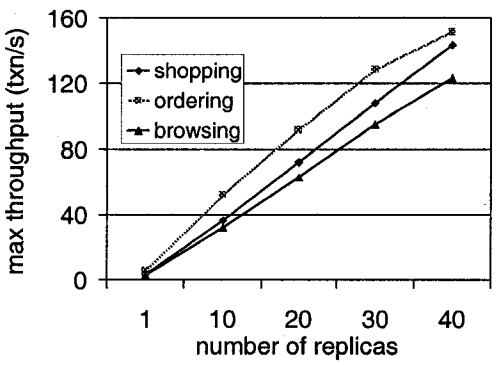


Figure 6.10: Scalability of SIMC with TPC-W

Scalability for browsing and shopping is basically linear up to 40 nodes. We only had 40 servers and therefore we do not have data beyond 40 replicas. For the ordering workload, the throughput increases linearly up to 30 replicas. However it starts to level off at this time point. At 30 replicas, the throughput is around 4.3 txn/s per replica, at 40 replicas, it is only 3.75 txn/s per replica. The reason is that updates have to be applied at all replicas. Although writeset application is faster than executing the entire SQL statement it takes resources from each replica that are no more available to execute further transactions.

6.5.5 Discussion

As a summary, SEQ and SIMC perform consistently better for update transactions than LP which suffers from an uneven distribution of requests. For read-only transactions, the performance is nearly the same for all protocols. SEQ performs slightly better than SIMC. However, this comes at the cost of less fault-tolerance since SIMC's usage of uniform reliable delivery assures that no transaction is lost. SIMC scales basically linearly in LANs. Considering that the performance difference between SEQ and SIMC is quite small, we would suggest that in a LAN SIMC is the best choice.

6.6 Wide area network

The previous section has shown that SIMC and SEQ perform better than LPnMsg in a LAN. Some of it is due to that LPnMsg has two message rounds for each operation in an update transaction. Most, however, is due to the uneven execution of transactions in LPnMsg. In a WAN, however, the extra message overhead of LPnMsg will likely have an extremely negative effect. Recall that SIMC and SEQ only require a constant number of messages per transactions. Thus, in this section we compare SIMC, SEQ and HYBRID not only against LPnMsg but also against LP1Msg which requires only two message rounds per transaction.

6.6.1 Experimental setup

We choose the shopping workload which has 20% write transactions in order to show the performance of both read-only and update transactions. For HYBRID, we used Spread [114] as group communication system.

We conducted our experiments in a WAN with 1-4 sites in Montreal (Canada), 1-3 sites in Madrid (Spain), 2 sites in Toronto (Canada), and 1 site in Edmonton (Canada).

All machines are PCs with similar computing power (e.g., AMD 1.5-3.0GHZ/0.5-2GB memory/Linux). The round trip times between machines in different clusters varies from 40 to 150 ms depending on the distances.

6.6.2 WAN without clusters: SEQ v.s. lazy primary copy

In this first scenario we compare SEQ against the two lazy primary copy approaches using 4 servers in 4 different cities. We show the results at the sequencer (Montreal) and at the non-sequencer that has the longest network distance from the sequencer (Madrid). We delay the analysis of SIMC to the next section.

We first analyze the CPU usage at the different servers since it has a quite large effect on the response time of the different algorithms. Fig 6.11.(a) shows the CPU usage at the primary server for LP1Msg and LPnMsg, and the sequencer for SEQ. As we have discussed in the LAN section, SEQ has a significant lower CPU usage than the lazy primary copy approaches, especially at high loads.

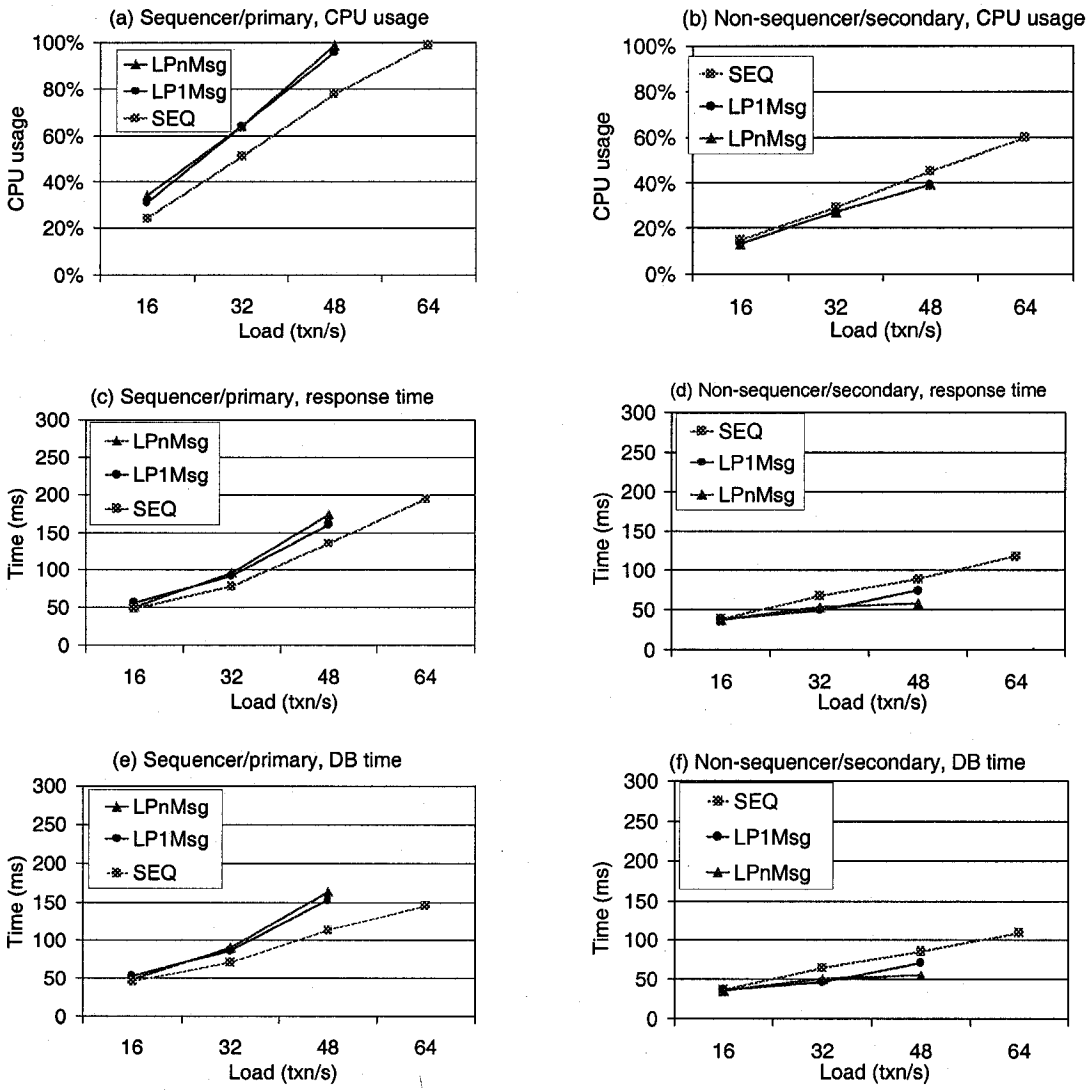


Figure 6.11: WAN without clusters: read-only transactions in shopping workload

With primary copy, both read and write operations of all write transactions are executed at the primary. In contrast, with SEQ, the read operations of write transactions submitted to non-sequencers are processed only at the non-sequencers, keeping the load at the sequencer lower, and LPnMsg has slightly higher CPU usage than LP1Msg because LPnMsg has to process more messages than LP1Msg. At the non-sequencers (Fig 6.11.(b)), SEQ has higher CPU usage than lazy primary copy for exactly the same reason that it distributes the load more evenly across the servers.

We now look at the average response times of read-only transactions submitted to either the sequencer/primary (Figure 6.11.(c)) or the non-sequencers/secondaries (Figure 6.11.(d)), and the time spent within the database (Figures 6.11.(e) and (f)). We can observe that these times are directly correlated with the CPU usage because read-only transactions do not have any communication overhead. Thus, SEQ has lower response at the sequencer than lazy primary copy at the primary (Fig. 6.11.(c)) and higher response time at the non-sequencers (Fig. 6.11.(d)). Furthermore, most of this response time is due to time spent in the database.

Let us now examine the behaviour of write transactions at the sequencer/primary. Figure 6.12.(a) and (c) show the average response time of update transactions and the time spent at the database, respectively. Write transactions submitted to the sequencer/primary are mainly affected by the time spent at the DB since there is no WAN communication. The DB time is directly correlated with the CPU usage (Figure 6.11.(a)). Thus, since the SEQ has the lowest CPU usage, it provides the shortest response times. LPnMsg and LP1Msg have similar response times since they have similar CPU usage.

Write transactions submitted to the non-sequencers/secondaries show a different picture. Figure 6.12.(b), (d), and (e) show average response time, time at the databases, and network time, respectively. Note that the y-axis scales to 1000 ms compared to 250 ms for the other figures. The response time of LPnMsg is four times higher than for LP1Msg and SEQ. The reason is that LPnMsg needs one WAN message round per operation (and in TPC-W an update transaction has on average four operations) while SEQ and LP1Msg only need one per transaction. Figure 6.12.(e) shows the time spent in the network. LPnMsg clearly has higher communication overhead than LP1Msg and SEQ. Figure 6.12.(d) shows that both LP1Msg and LPnMsg have higher DB overhead than SEQ. This is because the update transactions are executed at the primary database. We have seen before that the primary server in the lazy primary approaches has a higher CPU usage than the non-sequencers with SEQ, leading to longer execution times. Therefore, also LP1Msg has larger response times than SEQ at the non-sequencers.

We have also evaluated the bandwidth consumption since bandwidth usage is another crucial factor that has to be considered. At the primary , LPnMsg has the highest outgoing (Fig. 6.13.(a))and incoming (Fig. 6.13.(c)) bandwidth consumption because of the large number of messages needed.

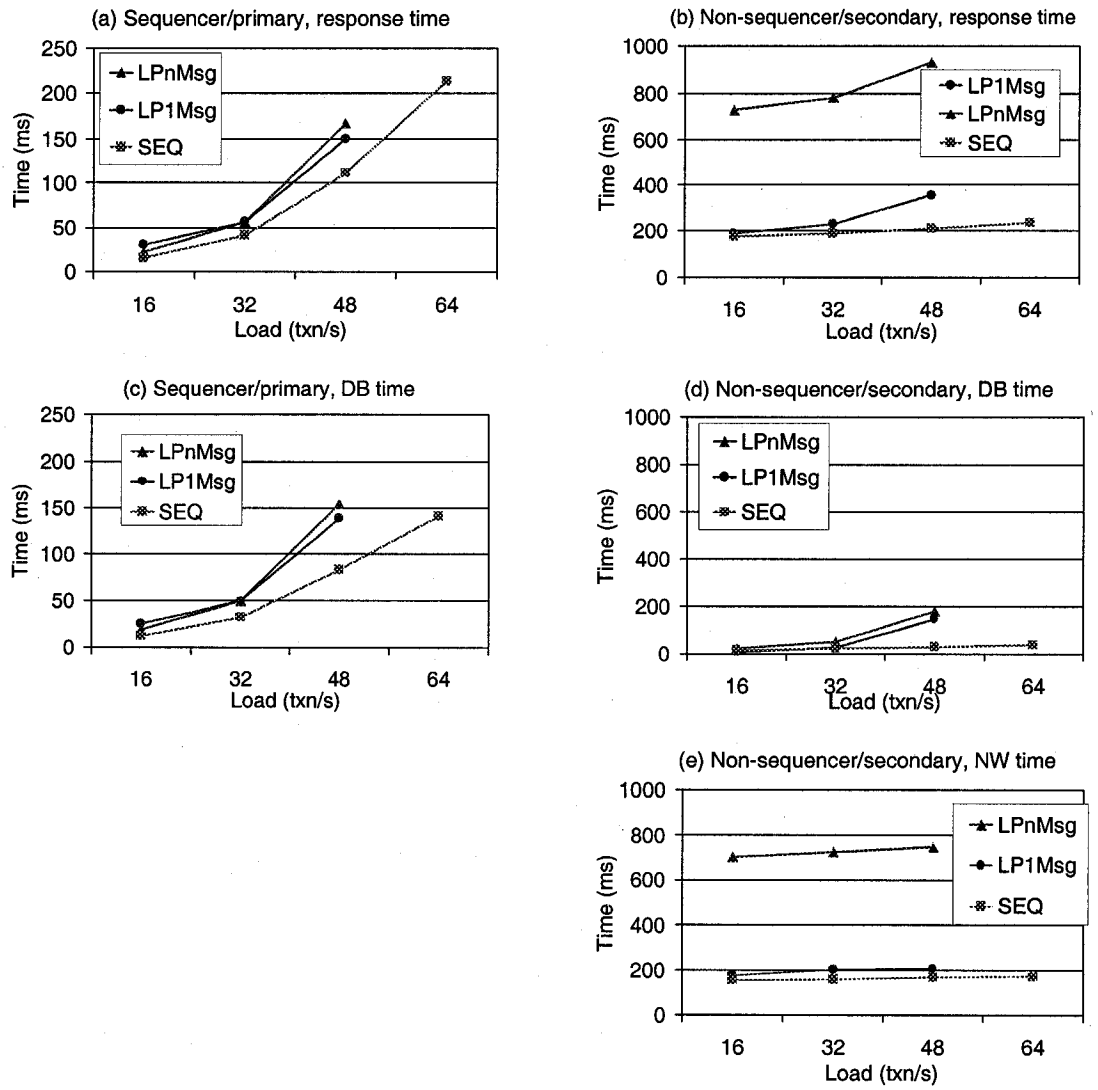


Figure 6.12: WANs without clusters: update transactions in shopping workload

LP1Msg has higher bandwidth consumption at the primary than SEQ at the sequencer because the primary must return query results of update transactions in LP1Msg but SEQ does not need to do so. The non-sequencer/secondary (Fig. 6.13.(b) and (d)) has similar tendency as the sequencer/primary.

Thus, we can summarize that SEQ by far outperforms LPnMsg, mainly because of message overhead. But it also outperforms LP1Msg. This is due to the more even distribution of load. Additionally, note that SEQ is more flexible than LP1Msg since it allows a standard JDBC interface

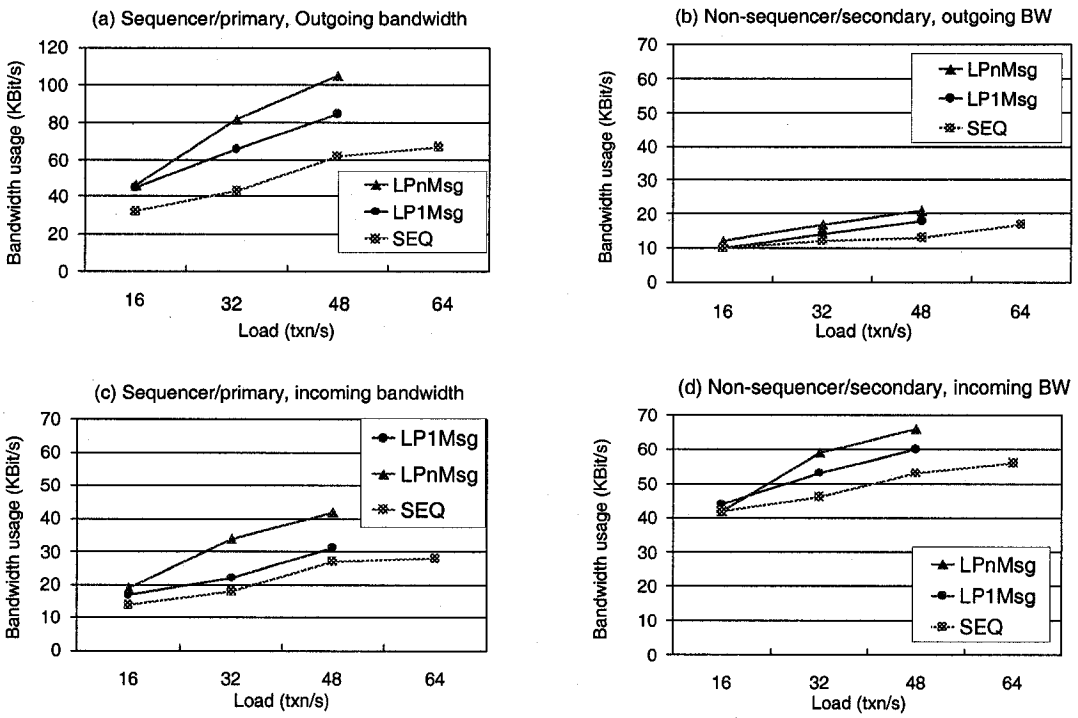


Figure 6.13: WANs without clusters: bandwith usage in shopping workload

without any further restrictions.

6.6.3 Overhead of GCS in WANs: SEQ v.s. SIMC

The previous section shows that SEQ is better than LPnMsg in WANs because of less communication overhead. SEQ only needs a constant number of message (i.e., one round trip) for a write transaction while LPnMsg needs several round trip messages. As SEQ, SIMC requires a constant number of message (i.e., one multicast message). It should have similar behaviour as SEQ just as in the LAN environment.

We know that SIMC requires total order multicast provided by GCSs. In Section 5.3.1 several total order algorithms have been discussed, such as the token-based algorithm in Spread [114], and the sequencer-based algorthm in JGroups [49]. They have different message latency which can have a tremendous effect on the performance in a WAN. In order to be fair to SIMC, we use the sequencer-based total order algorithm provided in JGroups. SEQ is actually derived from a sequencer-based

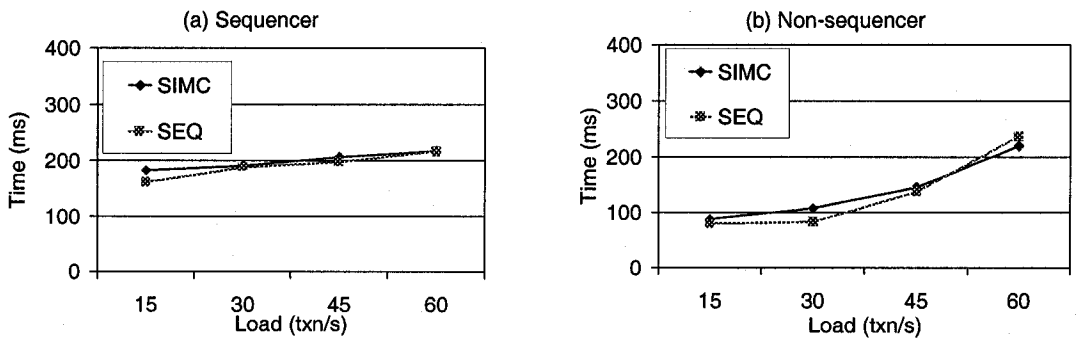


Figure 6.14: Overhead of GCS, SEQ v.s. SIMC, read-only transactions in shopping workload

total order algorithm. We would like to note that the sequencer-based total order algorithm of JGroups requires one and a half roundtrip messages per multicast, as has been discussed in Section 5.3.1. However, it does not provide uniform reliable delivery.

Figure 6.14.(a) and (b) show the average response times of read-only transactions at sequencer and non-sequencer replicas respectively. Response times are basically identical.

Figure 6.15.(a) and (b) show the average response times of write transactions at sequencer and non-sequencer replicas, respectively. At the sequencer replica (see Figure 6.15.(a)), SIMC has slightly larger response time than SEQ due to overhead of GCS. Additionally, SIMC requires a small network delivery time even at the sequencer replica (see Figure 6.15.(c)). Hence, SIMC has higher response time for write transactions at the sequencer replica.

At non-sequencer replicas (see Figure 6.15.(b)), SIMC also has larger response time than SEQ, mainly because of message delay. The difference of message delay between SIMC and SEQ is shown in Figure 6.15.(d). Moreover, not shown in the figures, SIMC has also higher CPU overhead, because the GCS is more CPU intensive than the socket communication in SEQ. The higher CPU load leads to larger DB time for SIMC compared to SEQ, which leads to the larger response time in SIMC.

We also conducted experiments using Spread which provides uniform reliable delivery. However, response times were always above 500 ms and clearly unacceptable.

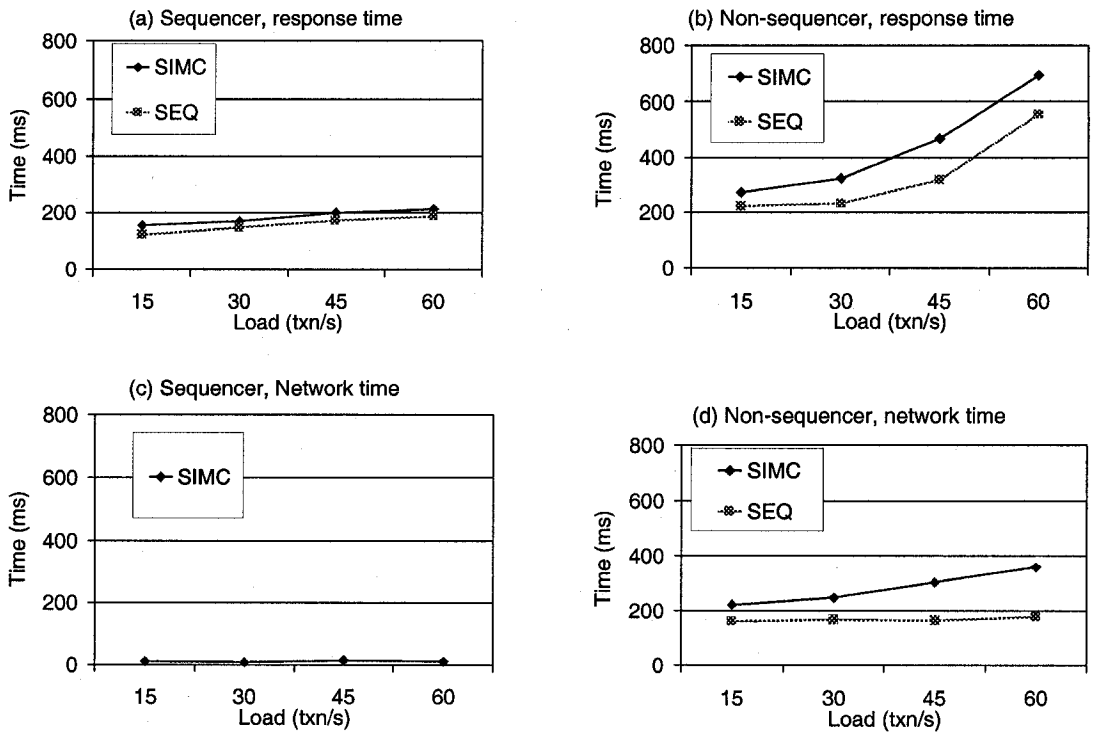


Figure 6.15: Overhead of GCS, SEQ v.s. SIMC, write txns in shopping workload

6.6.4 Clustered servers: HYBRID v.s. SEQ v.s. lazy primary copy

In this scenario we compare HYBRID against LP1Msg (being the better of the lazy primary copy protocols) and SEQ in the network topology shown in Fig 5.9. Recall that HYBRID provides a higher level of fault-tolerance than SEQ and LP1Msg. We study the results (1) at the global sequencer in the primary LAN, (2) at other replicas in the primary LAN, (3) at a local sequencer and (4) at other replicas in the secondary LANs.

Figure 6.16 shows the CPU usage ((a) and (b)), the average response time of read only transactions ((c) and (d)), and the time spent in the database ((e) and (f)) for sequencer and non-sequencers in the primary LAN. Figure 6.16 (c)-(f) show that the DB overhead is the main contributer to the response time of read-only transactions in the primary LAN. Figure 6.16.(a) shows that LP1Msg has the highest CPU usage at the primary in the primary LAN. Thus, LP1Msg has the largest response time for read-only transactions (see Figure 6.16.(c)). However, at the other replicas in the primary

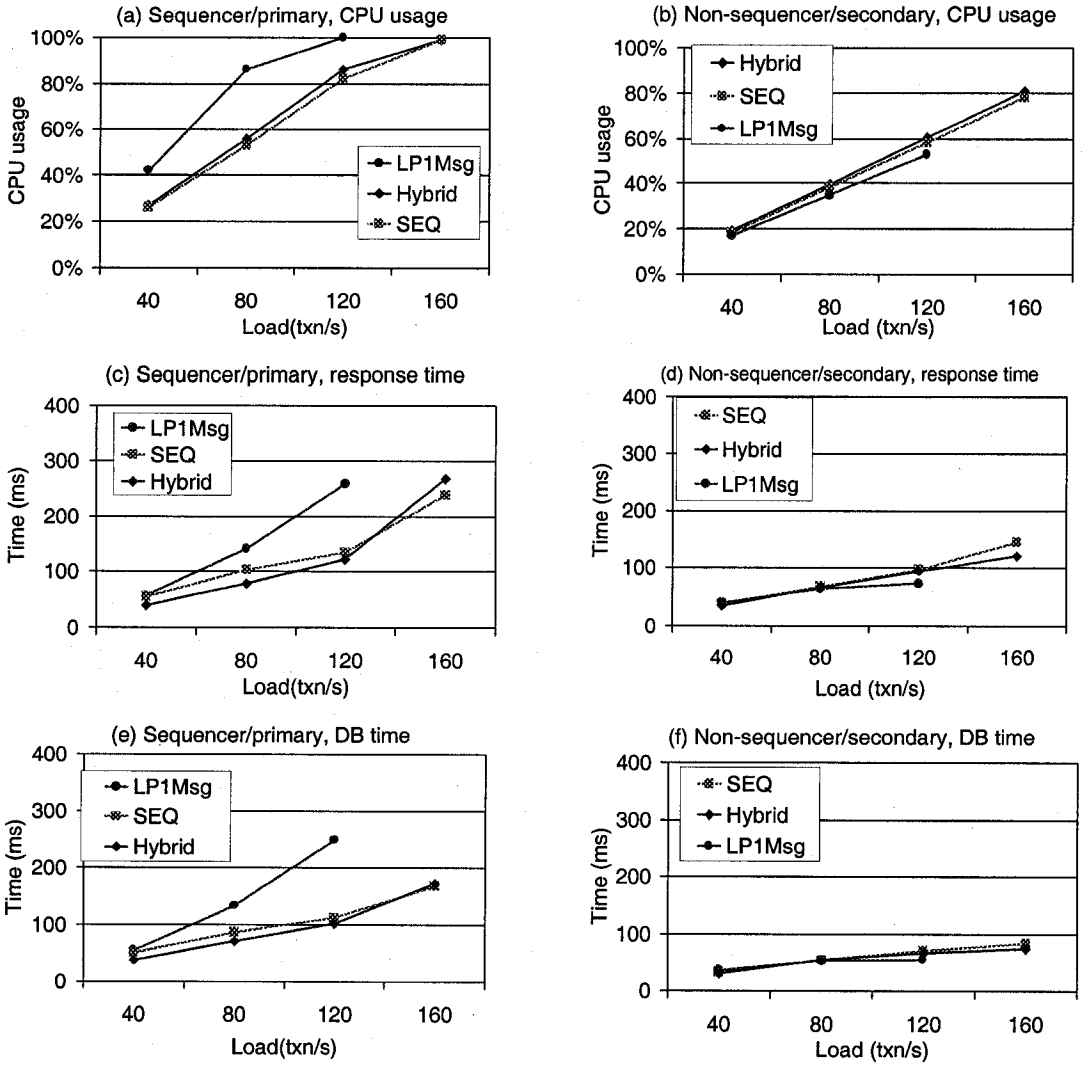


Figure 6.16: WANs with clusters: read-only transactions in primary LAN, shopping workload

LAN, Figure 6.16.(b) shows that LP1Msg has slightly lower CPU usage than HYBRID and SEQ, and thus slightly lower response times (see Figure 6.16.(d)). Comparing HYBRID with SEQ in Figure 6.16, HYBRID has slightly higher CPU usage due to the overhead of the GCS, but response times remain similar for read-only transactions.

Figure 6.17 shows the results of read-only transactions submitted to the replicas in secondary LANs. Figure 6.17 (a), (c), and (e) show CPU usage, average response time, and average DB time at a local sequencer, Figure 6.17 (b), (d), and (f) at a non-sequencer in a secondary LAN.

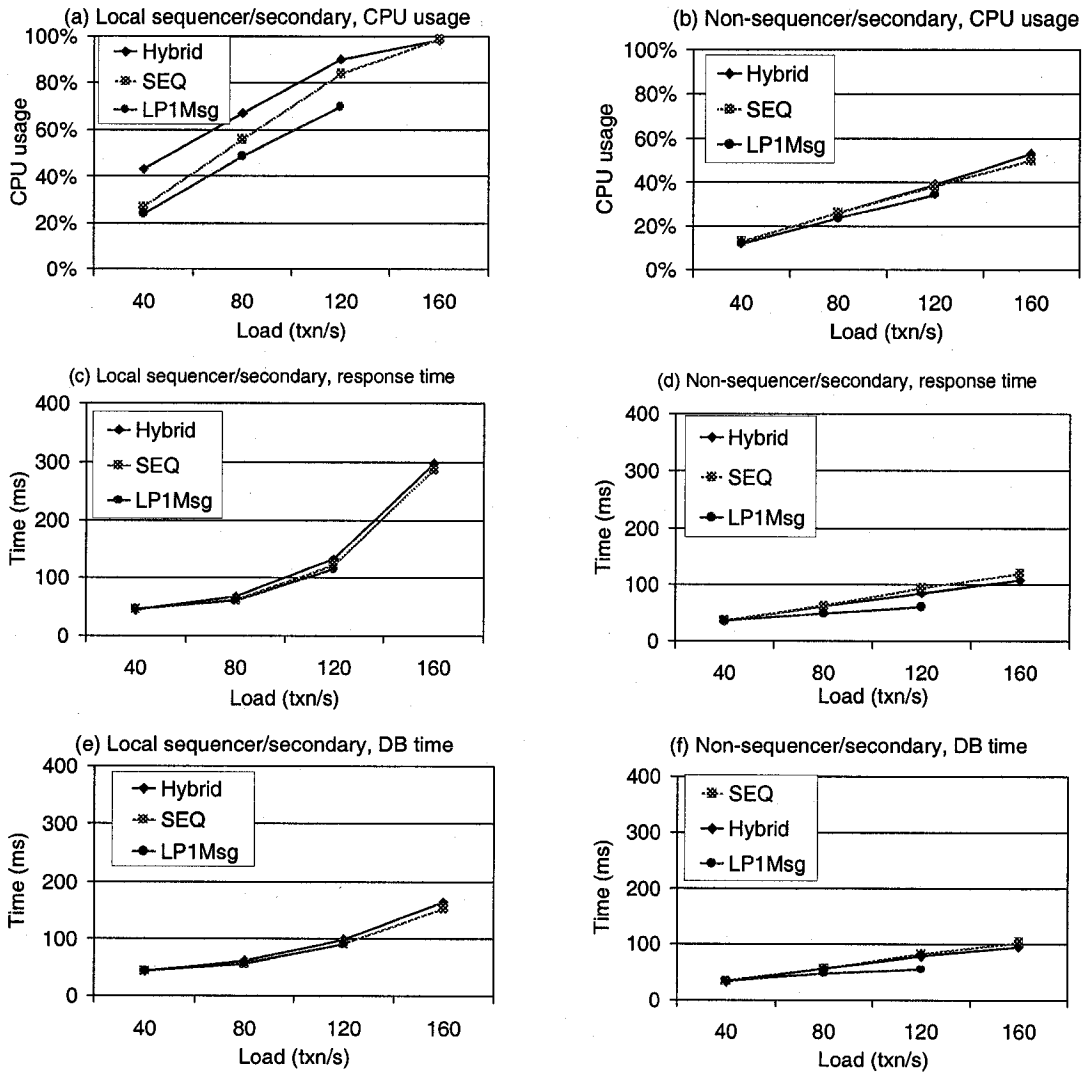


Figure 6.17: WANs with clusters: read-only transactions in secondary LANs, shopping workload

Although HYBRID has slightly larger CPU overhead at the sequencers, all protocols have very similar response times at all replicas. LP1Msg has slightly lower CPU overhead, and less response time at non-sequencers due to less load.

Figure 6.18 and 6.19 show the behavior of update transactions submitted to replicas in the primary and secondary LANs, respectively. There are figures for the average response time ((a) and (b)), the time at the DB ((c) and (d)), and at the network ((e) and (f)). Independently to which replica an update transaction is submitted, LP1Msg has the worst DB time and thus response times except

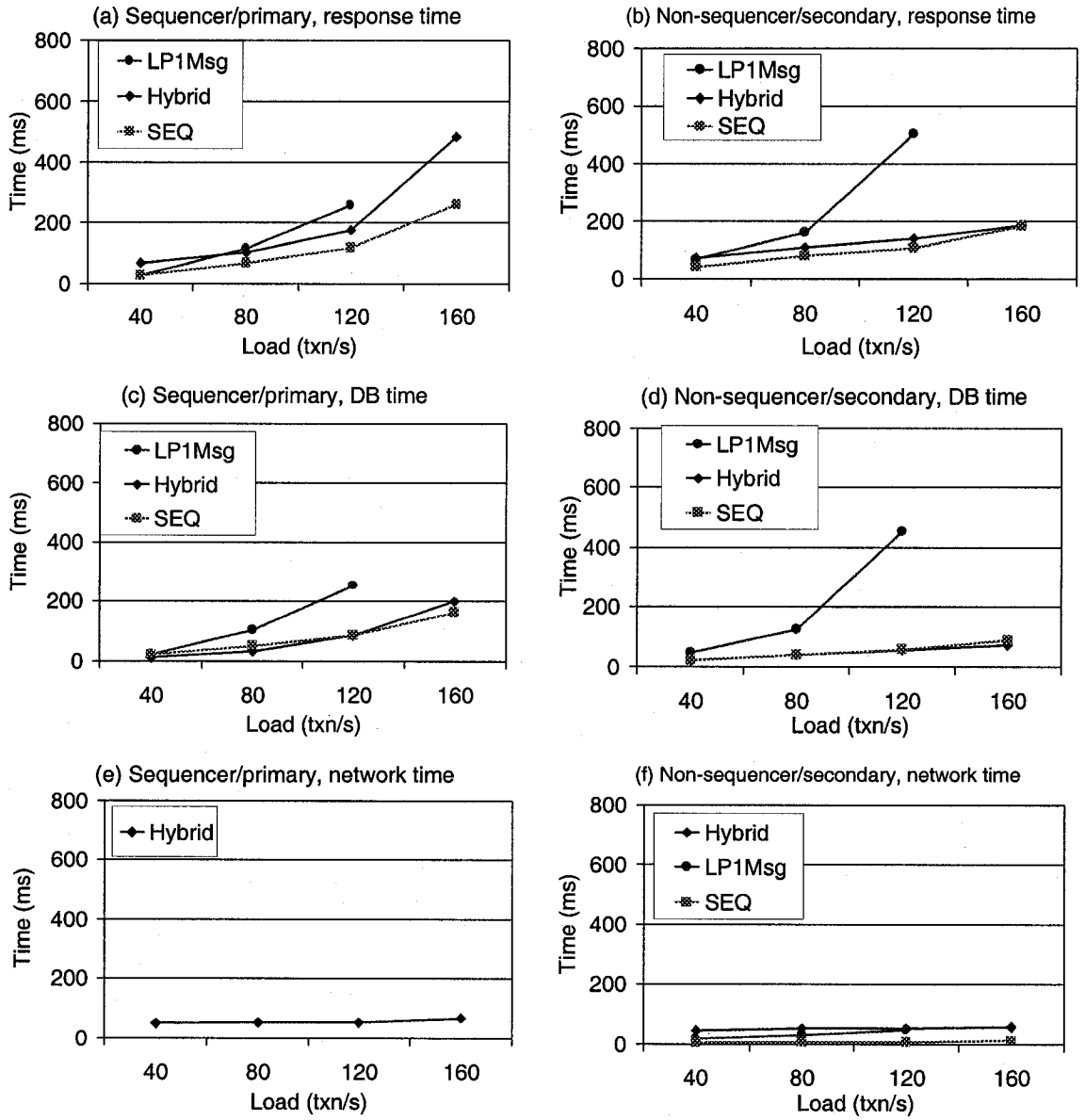


Figure 6.18: WANs with clusters: Update transactions in primary LAN, shopping workload

for transactions submitted to the primary replica at very low loads. The reason is that all operations of update transaction must be executed at the primary database. Comparing HYBRID with SEQ, both spend similar time in the DB. However, HYBRID has the additional cost of the GCS resulting in higher response times. However, the difference is relatively small (20-50 milliseconds or 15% in

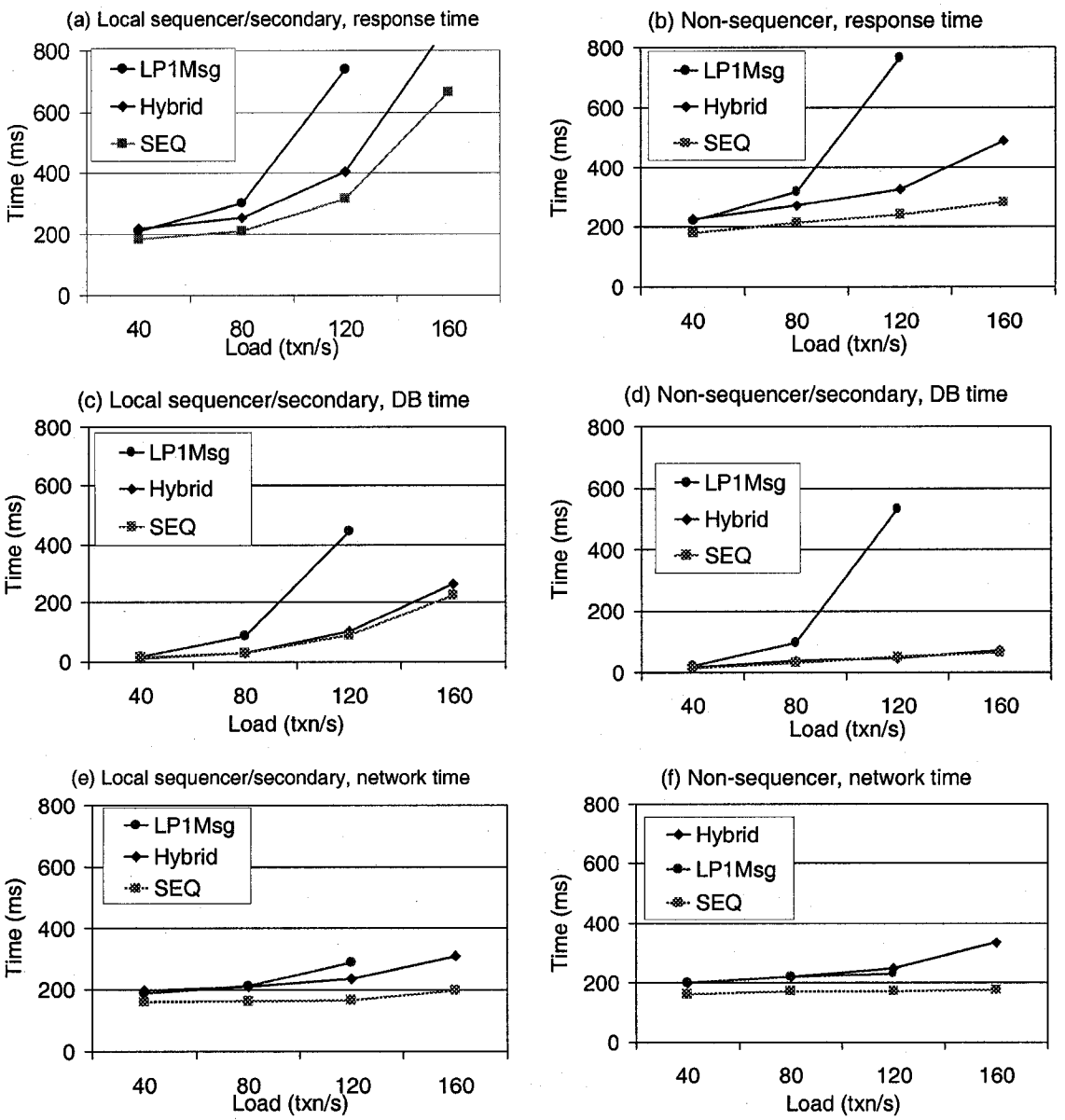


Figure 6.19: WANs with clusters: Update transactions in secondary LANs, shopping workload

most cases). This is the cost of stronger fault tolerance provided by HYBRID.

6.6.5 Discussion

In WANs, LPnMsg is much worse than SEQ because it requires two WAN message rounds per operation instead of two per transaction as in SEQ. Even with two WAN message rounds per transaction, LP1Msg is still worse than SEQ because of uneven read load distribution. Additionally, it has a much more resticted interface than SEQ.

SIMC provides more fault tolerance than SEQ. But uniform reliable multicast in WANs is very costly. Even without uniform reliable multicast, SIMC is still worse than SEQ because of the overhead of GCS. But in this case they have similar guarantee of fault tolerance. We conclude that SEQ outperforms SIMC in WANs.

HYBRID provides more fault tolerance than SEQ in WANs with clusters, with slightly higher cost than SEQ. It is because uniform reliable multicast in LANs is not as costly as in WANs. We suggest to use HYBRID in such an environment.

Chapter 7

Conclusions and future work

7.1 Summary

This thesis studies database replication in terms of correctness, performance, and practicability. It is motivated by the newly emerged isolation level SI used in commercial systems, and the fact that existing replica control algorithms perform badly in WANs and have many restrictions that make them difficult to use in practise.

7.1.1 New correctness criteria, 1-copy-SI and 1-copy-SI+IC

Snapshot Isolation (SI) is a new isolation level for transactions. It is weaker than serializabilty but more attractive because read and write operations do not block each other. There exist several replica control protocols based on SI for replicated systems. However, little has been done to formally describe what SI means in a replicated system. In Chapter 3, we propose a new isolation level, 1-copy-SI, based on Generalized Isolation Definition (GID) [3, 2]. Our formalism is convenient and straightforward to use. Moreover, it provides an implementation independent definition of 1-copy-SI. We discuss a set of necessary and sufficient conditions that make it easy to determine whether a history is 1-copy-SI and to show that a replica control mechanism provides 1-copy-SI.

Furthermore, we note that most existing protocols do not consider integrity constraints and thus, they do not work for databases with integrity constraints. In particular, we are not aware of any work

that considers integrity constraints in combination with SI. Hence, we propose a new isolation level SI+IC stronger than SI but weaker than Serializability (SE). A SI+IC history provides SI guarantees for read and write operations. Additionally, it respects integrity constraints. Based on SI+IC, we propose the corresponding correctness criterion, 1-copy-SI+IC, for replicated histories. We also discuss necessary and sufficient conditions that make it easy to determine whether a replicated history is 1-copy-SI+IC.

7.1.2 Performance

We analyze carefully the existing replica control protocols and find that most of them do not work well in WANs because of excessive number of messages within one transaction. Some of them require one roundtrip message in WANs for one operation. We propose a decentralied architecture and two protocols, SIMC and SEQ, that reduce the number of messages in WANs to be one multicast or one roundtrip per transaction. SIMC provides better fault-tolerance than SEQ by using uniform reliable multicast provided by GCS. However, SEQ has better response time. We also carefully discuss the fail-over procedure in SEQ. Both SEQ and SIMC have better load distribution potentials than lazy primary copy approaches, the most commonly used approach in commercial systems. This is because all update transactions must be performed at the primary replica. In constrast, in SIMC and SEQ, they are executed at the replica to which they are submitted.

To better utilize the network configuration, we propose a protocol HYBRID which combines SIMC and SEQ. HYBRID is designed for WANs with several clusters. It uses SIMC in its primary cluster but SEQ in its secondary clusters and in between primary and secondary clusters. HYBRID provides better fault tolerance guarantee than SEQ and better performance than SIMC. It is a tradeoff between SIMC and SEQ, and a practical choice from an engineering point of view.

7.1.3 Practicability

Many existing protocols have certain restrictions, allowing for a less flexible interface. Some update everywhere approaches require the knowledge of all tables to access in a transaction at start time. Our protocols do not have these restriction because they first execute the transaction at any replica

and then multicast the writeset to other replicas.

Additionally, many update everywhere approaches do not work for databases with integrity constraints. Integrity constraints are very important in databases. Our protocols handle integrity constraints by using the concurrency control module of the database system. Our protocols are also compatible to databases that implement SI using the first-updater-wins rule.

7.2 Future work

7.2.1 Enhancement to the integrity constraint model

We see two issues in regard to integrity constraints that could deserve more attention. Firstly, so far, we ignore that many database systems allow a CASCADE option. For instance, if a transaction wants to delete a department for which there are employees, instead of aborting, it also deletes the employees. Our integrity model needs to be enhanced to capture this behaviour.

Secondary, our protocols so far leave the checking of integrity constraints to the database replicas. This might result in executions at all replicas that will lead to abort. If we are able to check integrity constraints at the middleware layer, we might be able to develop a series of optimizations.

7.2.2 Partial replication and peer-to-peer databases

We have seen that SIMC can scale up to 40 replicas which is quite good already for enterprise applications. However, it is not suitable for peer-to-peer applications which require the support of thousands of nodes. This is a problem inherent to ROWA approaches because writes must be executed everywhere. To scale up to thousands of nodes, partial replication has to be considered. Partial replication is an essential functionablity in peer-to-peer systems. [77, 47, 32, 93, 30] study the replication problems in peer-to-peer systems but they mainly focus on replication at the granularity of files and put little focus on updates. Database replication in peer-to-peer systems is more challenging because it might require semantic interaction of data residing at different replicas. There exist several peer-to-peer databases such as PeerDB [86, 84], Piazza [53], and AmbientDB [19]. It would be interesting to see whether we could apply 1-copy-SI to partial replication in peer-to-peer

databases.

Furthemore, since WANs are getting faster and have larger bandwidth, it might make sense to retrieve data from the memory of other replicas over the network instead of from the local disk. In a peer-to-peer setup, we could fit a huge database into memories of thousands of replicas. Thus, we can reduce the I/O time of queries by visiting nearby replicas which have the data in their memories. [39] actually discusses the possiblity of achieving better scalability by reading data from memories at other replicas. However, consistency will play a huge role in this context.

7.2.3 Applying database replication to applications

Another option of future work is to apply database replication to existing distributed applications. There are many distributed applications requiring strong data consistency guarantee. [115, 107, 116] describe the benefits of applying database replication to *web services*. [74] discusses the possibility of applying database replication to *Massive Multi-player Online Game (MMOG)*. [79] discusses how to apply *distributed versioning*, a replica control algorithm, to transactional memory. [102, 31] discuss how to apply snapshot isolation to transactional memory. As a strong consistency level in replicated systems, 1-copy-SI could bring interesting properties and benefits to transactional memory.

Bibliography

[1] A. E. Abbadi, D. Skeen, and C. Cristian. An efficient fault-tolerant protocol for replicated data management. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, 1985.

[2] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, MIT, Cambridge, 1999.

[3] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, 2000.

[4] F. Akal, C. Turker, H. J. Schek, T. Grabs, and Y. Breitbart. Fine-grained lazy replication with strict freshness and correctness guarantees. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2005.

[5] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. On the performance of consistent wide-area database replication. Technical Report CNDS-2003-3, CNDS, John Hopkins University, 2003.

[6] Y. Amir and C. Tutu. From total order to database replication. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2002.

[7] C. Amza, A. L. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proc. of USENIX Annual Technical Conference*, 2003.

[8] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Int. Middleware Conference (Middleware)*, 2003.

[9] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 230–241, 2005.

[10] T. 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 (SIGMOD)*, 1998.

[11] ANSI. ANSI X3.135-1992, American National Standard for Information Systems Database Language SQL, November 1992.

[12] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 1995.

[13] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2006.

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

[15] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), December 1993.

[16] K. P. Birman. *Building Secure and Reliable Network Applications.* Prentice Hall, 1996.

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

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

[19] P. A. Boncz and C. Treijtel. AmbientDB: Relational query processing in a P2P network. In *Proc. of Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, pages 153–168, 2003.

[20] 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 (SIGMOD)*, 1999.

[21] 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)*, 1997.

[22] R. Carr. The tandem global update protocol. In *Tandem Systems Review*, June 1985.

[23] E. Cecchet, J. Marguerite, and W. Zwaenepoel. RAIDb: redundant array of inexpensive databases. In *Technical Report 4921 INRIA*, 2003.

[24] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proc. of USENIX Annual Technical Conference*, 2004.

[25] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2), 1996.

[26] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In *Proc. of the IEEE Int. Conf. on Autonomic Computing (ICAC)*, Dublin, Ireland, June 2006.

[27] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computer Surveys*, 26(2):145–185, 1994.

[28] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computer Surveys*, 33(4):427–469, December 2001.

[29] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, 1996.

[30] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proc. of Conf. on applications, technologies, architectures, and protocols for computer communications*, 2002.

[31] C. Cole and M. Herlihy. Snapshots and software transactional memory. *Science of Computer Programming*, 58(3), December 2005.

[32] F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen. Autonomous replication for high availability in unstructured P2P systems. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, 2003.

[33] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, 2004.

[34] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[35] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computer Surveys*, 17(3):341–370, 1985.

[36] X. Defago, A. Schiper, and P. Urban. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Transactions on Information and Systems*, E86-D(12), December 2003.

[37] A.Y. Dolev, D. Krameer, and S. Malki. Transis: A communication sub-system for high availability. In *Proc. of the IEEE Int. Conf. on Fault-Tolerant Computing Systems (FTCS)*, 1992.

[38] E. Pacitti and T. Özsu and C. Coulon. Preventive multi-master replication in a cluster of autonomous databases. In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, 2003.

[39] S. Elnikety, S. G. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proc. of European Conf. on Systems (EuroSys)*, 2006.

[40] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, 2005.

[41] Ensemble. Group communication systems, http://dsl.cs.technion.ac.il/projects/ensemble/.

[42] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[43] A. Fekete, N. A. Lynch, and A. A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Computer Surveys*, 19(2), 2001.

[44] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Record*, 33:12–14, 2004.

[45] S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. The Leganet system: Freshness-aware transaction routing in a database cluster. *Information Systems*, 32(2):320–343, 2007.

[46] D. K. Gifford. Weighted voting for replicated data. In *Proc. of ACM Symp. on Operating Systems Principles*, 1979.

[47] V. Gopalakrishnanand, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2004.

[48] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 1996.

[49] Java Groups. homepage: http://www.jgroups.org/.

[50] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[51] H. Guo, P. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: how to say Good Enough in SQL. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2004.

[52] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*. Addison Wesley, 1993.

[53] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(7):787–798, 2004.

[54] J. Holliday, D. Agrawal, and A. E. Abbadi. Partial database replication using epidemic communication. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2002.

[55] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group communication. In *Proc. of the IEEE Int. Conf. on Fault-Tolerant Computing Systems (FTCS)*, 1999.

[56] J. Holliday, D. Agrawal, and A. El Abbadi. Using multicast communication to reduce deadlock in replicated databases. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 196–205, October 2000.

[57] ISIS. Group communication systems, http://www.cs.cornell.edu/Info/Projects/ISIS/.

[58] P. Jalote. *Fault tolerance in distributed systems*. Prentice Hall, 1994.

[59] JavaGroups. homepage: http://www.jgroups.org/.

[60] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems (TODS)*, 28(3):257–294, September 2003.

[61] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving scalability of fault tolerant database clusters. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2002.

[62] K. Böhm and T. Grabs and U. Röhm and H. J. Schek. Evaluating the coordination overhead of replica maintenance in a cluster of databases. In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, 2000.

[63] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 1991.

[64] J. Kangasharju, J. Roberts, and K. Ross. Object replication strategies in content distribution networks. In *Computer Communications*, 2002.

[65] C. Karlsson, M. Karamanolis. Choosing replica placement heuristics for wide-area systems. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2004.

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

[67] 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)*, 2000.

[68] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*, Goteborg, Sweden, June 2001.

[69] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(4):1018–1032, 2003.

[70] G. M. Voelker L. Qiu, V. N. Padmanabhan. On the placement of web server replicas. In *Proc. of the IEEE Int. Conf. on Computer Communications (INFOCOM)*, 2001.

[71] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[72] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Consistent data replication: Is it feasible in WANs? In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, 2005.

[73] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Middleware based data replication providing snapshot isolation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2005.

[74] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Applying database replication to multi-player online games. In *Proc. of Annual Workshop on Network and Systems Support for Games (Netgames)*, 2006.

[75] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Enhencing edge computing with database replication. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, 2007.

[76] C. Liu, B. G. Lindsay, S. Bourbonnais, E. Hamel, T. C. Truong, and J. Stankiewitz. Capturing global transactions from multiple recovery log files in a partitioned database system. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2003.

[77] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of the Int. Conf. on Supercomputing*, 2002.

[78] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems (2nd edition)*. Prentice Hall, 1999.

[79] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[80] Microsoft SQL Server 2005 Replication. http://msdn2.microsoft.com/en-us/library/ms151198.aspx, 2007.

[81] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[82] F. D. Munoz-Esco, J. Pla-Civera, M. I. Ruiz-Fuertes, L. Irun-Briz, H. Decker, J. E. Armendariz-Inigo, and J. R. Gonzalez de Mendivil. Managing transaction conflicts in middleware-based database replication architectures. *srds*, 00:401–420, 2006.

[83] S. Navaratnam, S. T. Chanson, and G.W. Neufeld. Reliable group communication in distributed systems. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 1988.

[84] W. S. Ng, B. C. Ooi, K. L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, 2003.

[85] M. Nicola and M. Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, pages 645–672, 2000.

[86] B. C. Ooi, K. L. Tan, A. Zhou, C. H. Goh, Y. Li, C. Y. Liau, B. L., W. S. Ng, Y. Shu, X. Wang, and M. Zhang. PeerDB: Peering into personal databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2003.

[87] E. Pacitti, P. Minet, and E. Simon. Fast algorithm for maintaining replica consistency in lazy master replicated databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[88] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3):237–267, 2001.

[89] E. Pacitti, E. Simon, and R. N. Melo. Improving data freshness in lazy master schemes. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 1998.

[90] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Consistent database replication at the middleware level. *ACM Transactions on Computer Systems (TOCS)*, 23(4), November 2005.

[91] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, 1998.

[92] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14:71–98, 2003.

[93] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, load balancing and efficient range query processing in DHTs. In *Proc of Int. Conf. on Extending Database Technology (EDBT)*, 2006.

[94] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *Int. Middleware Conference (Middleware)*, 2004.

[95] C. Plattner, G. Alonso, and M. T. Özsu. Dbfarm: A scalable cluster for multiple databases. In *Int. Middleware Conference (Middleware)*, 2006.

[96] C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with satellite databases. *VLDB Journal, To appear*, 2007.

[97] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2003.

[98] 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, 1995.

[99] R. V. Renesse, K.P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[100] Oracle Replication. Oracle database advanced replication, 10g release 2 (10.2), 2005. http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14226/toc.htm.

[101] Sybase Replication. Replication strategies: data migration, distribution, and syncrhonization, 2003. A Sybased White Paper. http://www.sybase.com.

[102] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[103] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GlobData middleware. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[104] U. Röhm, K. Böhm, H-J. Schek, and H. Schuldt. FAS- a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2002.

[105] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.

[106] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computer Surveys*, 37(1), 2005.

[107] J. Salas, F. Perez-Sorrosal, M. Patiño-Martínez, and Ricardo Jiménez-Peris. WS-replication: a framework for highly available web services. In *Proc. of the Int. Conf. on World Wide Web (WWW)*, 2006.

[108] R. Schenkel and G. Weikum. Integrating snapshot isolation into transactional federations. In *Proc. of the Int. Conf. on Cooperative Information Systems (Coopis)*, September 2000.

[109] R. Schenkel, G. Weikum, N. Weissenberg, and X. Wu. Federated transaction management with snapshot isolation. In *Proc. of the Int. Workshop on Foundations of Models and Languages for Data and Objects*, 1999.

[110] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 2006.

[111] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *Proc. of European Conf. on Systems (EuroSys)*, 2006.

[112] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Network Computing and Applications*, 2001.

[113] JDBC specification. http://java.sun.com/products/jdbc, 2007.

[114] Spread. homepage: http://www.spread.org/.

[115] C. L. Sun, Yi Lin, and B. Kemme. Comparison of UDDI registry replication strategies. In *Proc. of Int. Conf. on Web Services (ICWS)*, 2004.

[116] M. Surgihalli and K. Vidyasankar. A lazy replication scheme for loosely synchronized UDDI registries. In *Proc of Int. Conf. on Parallel and Distributed Computing Systems (PDCS)*, pages 477–482, 2005.

[117] M. Szymaniak, G. Pierre, and M. V. Steen. Latency-driven replica placement. In *Proc. of the Int. Symp. on Applications and the Internet (SAINT)*, 2005.

[118] Transaction Processing Performance Council. TPC-W Benchmark. homepage: http://www.tpc.org/tpcw.

[119] Transaction Processing Performance Counil. homepage: http://www.tpc.org/.

[120] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2001.

[121] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in database and distributed systems. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2000.

[122] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(4):551–566, April 2005.

[123] WISC. PHARM, TPC-W Java implementation. homepage: http://mitglied.lycos.de/jankiefer/tpcw/index.html.

[124] S. Wu and B. Kemme. Postges-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, 2005.