A Paradigm for
Concurrency Control Protocols
for Distributed Databases

Amr El Abbadi
Ph.D. Thesis

87-853
August 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501
A PARADIGM FOR CONCURRENCY CONTROL PROTOCOLS
FOR DISTRIBUTED DATABASES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Amr El Abbadi

August, 1987
© Amr El Abbadi 1987

ALL RIGHTS RESERVED
A Paradigm For Concurrency Control Protocols For Distributed Databases

Amr El Abbadi, Ph.D.
Cornell University 1987

In this thesis, we present a paradigm for concurrency control protocols for distributed replicated databases. This paradigm presents a framework for both developing and analyzing concurrency control protocols, especially those that are designed to handle partitioning failures. Any concurrency control protocol that is an instance of the paradigm must be correct. We show that several known protocols are instances of this paradigm. Consequently, these seemingly unrelated protocols can now be compared and their understanding is simplified.

We also present two new concurrency control protocols: the virtual partitions protocol and the accessibility thresholds protocol. Both protocols allow the reading and writing of data in spite of site and communication failures, even when these failures lead to network partitioning. In neither protocol is it ever necessary for a read operation to physically access more than one copy, which makes these protocols desirable for applications where efficient read operations are necessary. The accessibility thresholds protocol provides the database designer with much flexibility in trading off the cost of executing operations and the availability of data objects. Unlike previous protocols, the cost of executing operations on an object is separated from the read and write availability of that object.
Biographical Sketch

Amr El Abbadi was born on November 3, 1958 in Cambridge, England, but spent most of his life in the more moderate climate of the Eastern Mediterranean. He attended schools in both Alexandria, Egypt and Beirut, Lebanon. He received his bachelor's degree in Computer Science and Automatic Control from Alexandria University in June 1980. He then spent a year as a demonstrator in the same department before crossing the Atlantic in August 1981 and enrolling in the Ph.D. program at Cornell University in Ithaca, New York. At Cornell, he was a teaching assistant during his first three years, and then a research assistant during the last three years there. In January 1984, he received his Master of Science degree. He married Janet E. Head on August 19, 1984 in Jacumba, California, and then again in La Habana, Cuba on January 25, 1985. In September 1987 he will join the University of California at Santa Barbara.
ال ابا و امی
Acknowledgements

First of all, I would like to thank my advisor Sam Toueg, for his constant guidance and assistance during the development of this work. His enthusiasm to discuss and develop vague and obscure ideas has been invaluable. His willingness to wade through endless drafts patiently and meticulously is unique. I am grateful to Ken Birman for his encouragement and for always making me see the bright side of things. Thanks also go to Dale Skeen who first launched me into the enterprise of partitioned databases. I would also like to express my appreciation to Juris Hartmanis and Barbara Lust for serving on my committee. Tommy Joseph and Thomas Raeuchle deserve special thanks for the many discussions we had together. I am also grateful to Gil, Koo and Ajei for reading various drafts of this thesis, and to Carolyne for supplying me with Tandem papers.

There are numerous people whom I would like to thank for their friendship and support, and of course for the surprise parties, video watching parties, dinner parties, and slide show parties. These thanks go to Aliza, Anne, Gil, Hilary, Jim, Kate, Koo, L.M., Micah, Rick, Rogerio, Solveig, Tommy and TK, without whom Ithaca would not have been the Ithaca I know and love.

I am especially grateful to my parents for their constant support and guidance even when an ocean separated us from each other. Finally, my thanks go to Janet without whom I would never have persevered to finish this thesis.
# Table of Contents

1. Introduction ........................................................................................................... 1

2. A Model for Distributed Databases ................................................................. 7
   2.1 Introduction ..................................................................................................... 7
   2.2 A Distributed System Model ......................................................................... 7
   2.3 A Distributed Database Model ....................................................................... 8
   2.4 Correctness Criteria ....................................................................................... 10
      2.4.1 Correctness Criteria for Non-replicated Databases .............................. 10
      2.4.2 Correctness Criteria for Replicated Databases .................................... 12
   2.5 Concurrency Control Protocols .................................................................... 14

3. The Virtual Partitions Protocol ........................................................................ 17
   3.1 Introduction ..................................................................................................... 17
   3.2 A Protocol for an Idealistic Network ............................................................ 18
   3.3 The Replica Control Protocol ......................................................................... 22
   3.4 View Management Protocol .......................................................................... 28
   3.5 Conclusion ...................................................................................................... 31

4. The Accessibility Thresholds Protocol ............................................................. 32
   4.1 Introduction ..................................................................................................... 32
   4.2 The Replica Control Protocol ......................................................................... 33
      4.2.1 Accessibility Thresholds, Quorums and User Transactions .................. 34
4.2.2 Update Transactions ........................................... 37
4.2.3 Policies for Changing Views .................................. 43
4.3 Proof of Correctness .............................................. 45
  4.3.1 Extensions to the Standard Serializability Theory .......... 45
  4.3.2 The Proof ..................................................... 47
    4.3.2.1 Construction of $I$-$SG[L]$ .......................... 47
    4.3.2.2 Acyclicity of $I$-$SG[L]$ .............................. 52
4.4 Optimizations .................................................... 55
  4.4.1 Increasing Data Availability ................................ 55
    4.4.1.1 Relaxing the Read and Write Rules ................. 55
    4.4.1.2 Using Multi-Versions ................................. 59
4.4.2 Optimizing Update Transactions .............................. 63
  4.4.2.1 Reducing the Number of Update Transactions ............ 63
  4.4.2.2 Reducing the Granularity of Update Transactions .......... 64
  4.4.2.3 Reducing Communication Costs of Update Transactions ... 65
  4.4.2.4 Optimizing Update Transactions after Site Failures .... 66
4.5 Comparison with Other Work .................................... 67
4.6 Commercial Distributed Databases ................................ 72
4.6 Conclusion ....................................................... 73
5. The Group Paradigm ........................................................................... 75

5.1 Introduction ................................................................................. 75

5.2 A Simple Motivating Example ...................................................... 77

5.3 The Paradigm ............................................................................. 79

5.3.1 Groups .................................................................................... 80

5.3.2 Group Concurrency Control Protocols ...................................... 81

5.4 Serialization Group Model ........................................................... 83

5.5 Analysis of Protocols ................................................................. 90

5.5.1 The Basic Quorum Protocol .................................................... 91

5.5.2 The Missing Write Protocol ..................................................... 92

5.5.3 Class Conflict Protocol .......................................................... 94

5.5.4 The Virtual Partitions Protocol ................................................ 96

5.5.5 The Accessibility Threshold Protocol ....................................... 99

5.5.6 Herlihy's Generalized Quorum Protocol ................................... 103

6. Conclusion .................................................................................... 109

References ....................................................................................... 113
List of Figures

3.1. The View Management Protocol .............................................. 29

4.1. initiate_new_view procedure ................................................. 39

4.2. install_view procedure ...................................................... 40

4.3. update transaction .......................................................... 40

4.4. Response to access request ............................................... 42

4.5. inherit_view procedure ..................................................... 42
CHAPTER 1

Introduction

A distributed system is a set of sites connected by communication links. A site usually contains a processing unit and a storage device, and sites communicate with each other by sending messages through the communication links.

A database is a collection of objects. A user may access the database by executing a transaction, which is a sequence of read and write operations on objects. A transaction is guaranteed to take the database from one "consistent" state to another. If a set of transactions is executed concurrently, the result should be the same as that generated when the same set is executed in some sequential order. This notion is known as serializability [Bern87b, Gray78, Papa86, Ullm82]. A distributed database is a distributed system with a database in which different objects may reside on different sites. Furthermore, users may initiate transactions from any of the sites in the database.

Fault-tolerance is a major concern that arises when designing a distributed database. In a fault-tolerant distributed database, a transaction may read or write objects even if some sites and communication links fail. In general, the failure of a component may range from merely crashing [Hadz84], to performing arbitrary, possibly malicious actions [Peas80]. In this thesis, we restrict ourselves to the following types of failures: a site may fail by crashing, or by failing to send or receive messages [Perr86], and a communication link may fail by delaying or failing to
deliver messages [Hadz84].

To increase fault-tolerance, database objects are often replicated on different sites. If an object is implemented by several copies that reside on different sites, some copies of that object may still be available to a transaction, even after failures occur. Such replication, however, has its costs. The database designer must ensure that the user perceives a replicated object as a non-replicated one. Formally, this is the requirement that all transaction executions are one-copy serializable [Bern87b]. A concurrency control protocol is one that ensures one-copy serializability.

An example of a simple concurrency control protocol for replicated databases is the "read-one write-all" protocol. It is based on the following rule: a write operation writes all copies of an object, and a read operation reads any one copy. Such a protocol, however, does not allow a transaction to write objects that have copies on failed sites. To overcome this problem, Bernstein and Goodman [Bern84, Good83] proposed the available copies protocol, where write operations write only the copies on operational sites, and hence, can be executed despite site failures. In this protocol, special exclude and include transactions are executed to inform sites of the failure and recovery of sites. The protocol ensures one-copy serializability and tolerates site failures.

In addition to site failures, a distributed database system may also suffer from link failures. Site and link failures may lead to partitioning of the database, such that sites within a partition can communicate with each other, but not with sites in other partitions. The available copies protocol does not guarantee one-copy ser-
alizability when partitioning occurs. As an example, consider the case of a network divided into two partitions. The sites in each partition assume that all sites in the other partition have failed. Hence, the available copies protocol allows sites in both partitions to execute transactions that read and write the same object (as long as some copy of this object can be found in both partitions). Such transactions result in not one-copy serializable executions.

A simple solution to the problem of maintaining one-copy serializability in the presence of partitioning is Gifford's quorum protocol [Giff79]. With this protocol, an object may be written by writing a write quorum of copies, and not necessarily all the copies of the object; similarly, an object may be read by accessing a read quorum of copies. For each object, a write quorum of copies must intersect with any other read or write quorum of copies. Thus any read or write operation on an object is always "aware" of all previous write operations on that object. Gifford's protocol ensures one-copy serializability, even in the presence of site and link failures that lead to partitioning. One disadvantage of this protocol is that read operations must access more than one copy, even if no failures have occurred. This may not be desirable for applications where efficient read operations are necessary. Furthermore, with this protocol, the sum of the read and write quorums of any object must be greater than the number of copies of that object. Thus the cost of reading and writing an object increases as the number of copies increase. Hence, improving fault-tolerance by increasing the number of copies, directly increases the cost of executing operations.
In this thesis, we develop two new concurrency control protocols: the *virtual partitions protocol* and the *accessibility thresholds protocol*. Both protocols tolerate site and link failures, even when these failures lead to partitioning. Furthermore, they do not exhibit the disadvantages associated with the protocol in [Giff79]. In fact, they never require more than one copy of an object to be read even if the database partitions. Moreover, the sum of the read and write quorums of any object is allowed to be less than the number of copies of that object. This decreases the cost of operations.

The virtual partitions protocol uses information about the communication network to determine whether it is safe to access an object, and how to execute read and write operations. If the network partitions, a write operation that is allowed in a partition writes all the copies in that partition, a read operation accesses only one copy. This protocol was the first protocol developed to allow read operations to access only one copy even if the database partitions. The virtual partitions protocol is simple, but it requires an expensive two phase protocol to coordinate the views different sites have of the communication network.

In contrast to the virtual partitions protocol, the accessibility thresholds protocol does not require a separate two phase protocol for coordinating views. The accessibility thresholds protocol separates the issue of data availability from operation costs. In fact, the sum of the read and write quorums of any object is independent of the number of copies of that object. The protocol even allows the reading and writing of an object by touching a single copy of that object. Thus increasing availability by replication does not necessarily increase the cost of operations. The
protocol also provides the database with a large degree of flexibility in deciding the degree of data availability, as well as the cost of accessing data: A database designer first chooses the availability level of each object, and then chooses the cost of each operation on that object. Furthermore, the accessibility thresholds protocol attains maximum availability according to a measure of data availability defined by Coan et al. [Coan86].

From our study of these concurrency control protocols, we have extracted a paradigm that presents a unifying framework for describing and proving the correctness of concurrency control protocols. The paradigm is presented as a set of conditions, and any protocol that satisfies these conditions is considered an instance of the paradigm. We prove that any concurrency control protocol that is an instance of the paradigm ensures one-copy serializability. We show that both the virtual partitions and the accessibility thresholds protocols, as well as several other known protocols [Giff79, Eage83, Skee84, Herl87] are all instances of this paradigm. The paradigm simplifies the analysis and comparison of seemingly unrelated concurrency control protocols.

One advantage of our paradigm is the simplicity with which concurrency control protocols can be proven correct. To demonstrate this, we first prove the correctness of the accessibility thresholds protocol using the standard database model of Bernstein and Goodman [Bern87a, Bern87b]. This proof is quite complex. We then present a simpler proof by showing that the protocol is an instance of our paradigm.
This thesis is divided into six chapters. In Chapter 2 we present a standard database model that is based on logs and serialization graphs. We formally define serializability, one-copy serializability, and concurrency control protocols. In Chapter 3, we present the virtual partitions protocol. The goal of this chapter is to provide intuition for the rest of the thesis, and hence, we give a simple and informal description of the protocol, and we merely outline the proofs of correctness. The accessibility thresholds protocol is developed and formally presented in Chapter 4. We rigorously prove the protocol correct using serialization graphs, and compare its performance with other existing protocols. Finally, in Chapter 5, we develop the group paradigm and show that the best known protocols designed for executing transactions in a database that may partition are all instances of the paradigm. We conclude the thesis by summarizing our results and discussing future work.
CHAPTER 2

A Model for Distributed Databases

2.1. Introduction

In this chapter we present a model for databases in a distributed system. We first describe our model of a distributed system and of site and communication failures. We then present our model of a database, which is essentially the model developed by Bernstein and Goodman [Bern87a, Bern87b]. We define an object as an abstract data type that supports read and write operations, and a transaction as a partially ordered sets of operations issued by a user. A transaction appears to the user as if it were executed atomically; that is, as an indivisible entity on the database. To increase database availability, operations from different transactions may be allowed to interleave. We discuss the correctness criteria for the execution of a set of transactions first for non-replicated databases, where each object is implemented by a single copy, and then for replicated databases, where each object is implemented by several copies. We conclude with a brief discussion on concurrency control protocols that ensure serializability in both non-replicated and replicated databases.

2.2. A Distributed System Model

We consider a set of sites that are connected by bidirectional communication links. Associated with each site is a unique site-identifier, where site-identifiers form a total order. Sites communicate by message passing through the
communication links. We make no assumptions concerning the connectivity of the network or the details of the physical interconnection of the sites, although we assume the existence of a routing protocol [Tane81] that will successfully deliver messages between sites if possible.

We consider a network that is prone to both link and site failures. Sites may fail by crashing [Hadz84] or by failing to send or receive messages [Perr87]. Links may fail by crashing, or failing to deliver messages [Hadz84]. Furthermore, performance failures may delay the delivering of some messages. An important consequence of site and link failures is partitioning [Davi85, Skee82, Wrig83]. Sites in a partition can communicate with each other, but not with sites in other partitions. Link failures as well as the failure of gateway sites may cause partitioning failures.

2.3. A Distributed Database Model

A database consists of a set of objects. For example, in a bank database the different objects in the database represent the various accounts. In a distributed database, objects may reside on different sites. Each object has a value, and the state of the database is defined to be the values of all the objects in the database [Bern87b, Eswa76, Gray78]. Depending on the application, some states of the database are defined to be consistent. In a bank database, accounts with positive balances may be considered consistent (note there may be other constraints that have to be maintained between different objects of the database such as the sum of the checking and savings accounts must be greater some minimal value).
Each object defines two operations: the *read* operation, and the *write* operation. For object $x$, *read* $x$ returns the current value of $x$, and *write* $x$ assigns a new value to $x$. We assume that each object ensures the *atomic* execution of a single read or write operation; that is if two operations are executed on an object, one operation must completely terminate before the other operation is allowed to begin.

A transaction $t_i$ is a partially ordered set $(S_i, <_i)$, where $S_i$ is the set of all operations executed by $t_i$, and $<_i$ reflects the order in which they should be executed. Read and write operations executed by transaction $t_i$ on object $x$ are denoted $r_i[x]$ and $w_i[x]$. Each transaction is assumed to be *correct*; that is, if a transaction begins in a consistent state, and is executed alone, it leaves the database in a consistent state on termination. Returning to the bank database, we assume that no transaction, if executed alone, will leave a negative balance in an account. To ensure that the database is initially in a consistent state, we assume that an initial transaction $t_{init}$ is executed that writes all objects in the database before any other transaction.

In a distributed database a transaction may be initiated at any sites; this site is called its *initiator*. A read or a write of object $x$ is executed by forwarding the operation to the site where $x$ is stored, and the result of the operation is returned to the initiator of the transaction.

The database must ensure that when a transaction is executed, it is either executed in its entirety, or not at all. Hence, before a transaction terminates it either *commits* or *aborts* all changes it made to the database. The two phase commit protocol [Skee82, Gray78] ensures atomic commitment, but *blocks* if site
failures occur. A protocol is said to be \textit{blocking} if in some cases the execution of the protocol may not be able to terminate until failed sites have recovered. The three phase commit protocol [Skee82, Cheu85, Dwor83] is a non-blocking atomic commit protocol that tolerates site failures. Skee [Skee82] proves that if a network partitioning occurs during the execution of a transaction, no atomic commit protocol can guarantee the termination of that transaction (as long as the partition persists), and hence, partitioning failures may prevent some transactions from terminating. (For a detailed analysis of commit protocols in the presence of partition failures see [Skee82, Cheu86, Chun87].) In this thesis we do not address the problem of atomic commitment. Rather, our goal is to increase the availability of data after site and communication failures even when such failures lead to network partitioning.

2.4. Correctness Criteria

The execution of operations in the system is modeled by a log. A log $L$ over a set of transactions $T = \{t_1, t_2, \ldots, t_n\}$ is a partial order $(S, <_L)$ where $S$ is the set of all operations executed by transactions in $T$, and $<_L$ reflects the order in which the operations were executed.

2.4.1. Correctness Criteria for Non-replicated Databases

We first consider non-replicated databases where each object $x$ resides on a single site. Let $L$ be a log over a set of transactions $T$. Transaction $t_j$ \textit{reads} $x$ \textit{from} transaction $t_i$ in $L$ if:

1. $w_i[x]$ and $r_j[x]$ are in $L$, 

2. $w_i[x] \prec_L r_j[x]$, and

3. there is no $w_k[x]$ such that $w_i[x] \prec_L w_k[x] \prec_L r_j[x]$.

Two logs $L_1$ and $L_2$ are equivalent if for all $t_i, t_j$ and $x$, $t_j$ reads $x$ from $t_i$ in $L_1$ if and only if $t_j$ reads $x$ from $t_i$ in $L_2$. A serial log is a totally ordered log such that for every pair of transactions $t_i$ and $t_j$, either all operations executed by $t_i$ precede all operations executed by $t_j$, or vice versa. Since each transaction is assumed to be correct and preserve database consistency, a serial execution must also preserve database consistency. A log is serializable if it is equivalent to a serial log over the same set of transactions [Eswa76, Gray78]. Since a serial execution of transactions preserves the consistency of the database, we consider serializability as our correctness criterion. The problem of determining whether an arbitrary log is serializable is NP-complete [Papa79]. Here, we consider a subset of the class of serializable logs, the CP-serializable logs as defined below.

Two operations conflict if they both operate on the same object, and one of them is a write. A log $L$ is CP-serializable [Hadz84, Papa79] if there exists some serial log $L_s$ over the same set of transactions, such that if $op_1$ and $op_2$ conflict and $op_1 \prec_L op_2$, then $op_1 \prec_L op_2$. Note that $L$ is equivalent to $L_s$, and since $L_s$ is serial, $L$ is serializable.

A serialization graph $SG[L]$ of a log $L$ is a directed graph whose nodes are transactions and whose edges are: \( \{ t_i \rightarrow t_j \} \) if $\exists op_i$ executed by $t_i$ and $op_j$ executed by $t_j$ such that $op_i$ conflicts with $op_j$ and $op_i <_L op_j$. $L$ is CP-serializable if and only if $SG[L]$ is acyclic [Eswa76, Hadz84].
2.4.2. Correctness Criteria for Replicated Databases

We now consider replicated databases where each object is implemented by a set of copies that reside on different sites. The copy of object $x$ that resides on site $p$ is denoted by $x_p$. Each copy $x_p$ has a version number which is initialized to 1 by $t_{init}$. The set of all copies of object $x$ is denoted $\text{copies}[x]$, and the set of sites on which $x$ resides is denoted $\text{sites}[x]$. The number of copies of $x$ in the system is $n[x]$ ($n[x] = |\text{copies}[x]|$). In a non-replicated database all objects are implemented by a single copy.

An operation issued by a transaction on an object is called a logical operation. Such an operation is executed by a set of physical operations on the copies of the object. A logical write $w_i[x]$ is executed by

1. selecting a set of copies of $x$,
2. determining $\text{unmax}$, the maximum version number of the selected copies, and
3. writing all the selected copies and updating their version numbers to $\text{unmax} + 1$.

A logical read $r_i[x]$ is executed by

1. selecting a set of copies of $x$,
2. accessing all the selected copies to find the one with the highest version number, and
3. reading this copy.

The read, access and write operations that are executed on a copy $x_p$ by transaction $t_i$ are denoted $r_i[x_p]$, $a_i[x_p]$ and $w_i[x_p]$. 
A replicated database log \( L \) contains physical operations on the copies of objects. For each logical write \( w_i[x] \), there is a set \( \{x_p, \ldots, x_q\} \), the set of copies written, such that \( L \) contains \( w_i[x_p], \ldots, w_i[x_q] \). For each logical read \( r_i[x] \), there is a set \( \{x_p, \ldots, x_j, \ldots, x_q\} \), the set of copies accessed, such that \( L \) contains \( a_i[x_p] <_L r_i[x_j], \ldots, a_i[x_j] <_L r_i[x_j], \ldots, a_i[x_q] <_L r_i[x_j] \) (where \( x_j \) is the value read). Transaction \( t_j \) reads \( x \) from \( t_i \) if there is a copy \( x_p \) of object \( x \) such that:

1. \( w_i[x_p] \) and \( r_j[x_p] \) are in \( L \),
2. \( w_i[x_p] <_L r_j[x_p] \), and
3. There is no \( w_k[x_p] \) such that \( w_i[x_p] <_L w_k[x_p] <_L r_j[x_p] \).

Two logs \( L_1 \) and \( L_2 \) are equivalent if for all \( t_i, t_j \) and \( x, t_j \) reads \( x \) from \( t_i \) in \( L_1 \) if and only if \( t_j \) reads \( x \) from \( t_i \) in \( L_2 \). A log is one-copy serializable [Bern83, Bern87b] if it is equivalent to a serial log over the same set of transactions executed over a non-replicated database.

We extend the definition of conflict to both logical and physical operations. Two logical (physical) operations logically (physically) conflict if they both operate on the same object (copy) and one of them is a write. We also say that two transactions logically (physically) conflict if they both issue logically (physically) conflicting operations.

The serialization graph \( SG[L] \) of a replicated database log \( L \) is a directed graph whose nodes represent transactions, and whose edges are: \( \{t_i \rightarrow t_j|\exists \ op_i \ executed \ by \ t_i \ and \ op_j \ executed \ by \ t_j \ such \ that \ op_i \ physically \ conflicts \ with \ op_j \ and \ op_i <_L op_j \} \). The graph \( SG[L] \) orders transactions that issue physically conflicting operations in \( L \). However, two transactions may issue two logically conflicting
operations that may not physically conflict, and hence are not ordered by $SG[L]$. To order transactions that issue logically conflicting operations, we extend $SG[L]$ into a one-copy serialization graph, $1 - SG[L]$ by adding enough edges such that:

1. For each object $x$, $1 - SG[L]$ embodies a total order over all transactions that write $x$. This total order is called a write order for $x$, and is denoted $\Rightarrow_x$.

2. For each object $x$ and transactions $t_i$, $t_j$, $t_k$ such that $t_j$ reads $x$ from $t_i$ and $t_i \Rightarrow_x t_k$, $1 - SG[L]$ contains a path from $t_j$ to $t_k$. This path is called a reads-before path.

In [Bern83, Bern87a], Bernstein and Goodman prove that a log $L$ is one-copy serializable if $L$ has an acyclic $1 - SG[L]$ graph.

2.5. Concurrency Control Protocols

A concurrency control protocol is responsible for ensuring that transaction executions maintain the consistency of the database. In a non-replicated database, a trivial concurrency control protocol is one that allows only one transaction to be executed at a time. This ensures a serial execution, and hence correctness.

The goal of concurrency control is not only to ensure correctness, but also to increase the availability of the database. Therefore, a concurrency control protocol should allow the interleaving of operations from different transactions as long as serializability is ensured. A conflict preserving concurrency control protocol is one that ensures CP-serializability. The problem of developing and analyzing conflict preserving concurrency control protocols for non-replicated databases has been extensively studied [Eswa76, Gray78, Bern81], and several well known protocols have been proposed, such as the two phase locking protocol [Eswa76] and the
timestamp protocol [Reed83].

In a replicated database, the different copies of the object must behave as a single copy object, and all transaction executions must be one-copy serializable. For this reason, concurrency control protocols for replicated databases usually consist of two components. The first component is a concurrency control protocol for non-replicated databases [Gray78, Eswa76, Reed83]. The second component of the concurrency control protocol, the replica control protocol, translates logical operations to physical operations in a way that ensures the database is one-copy equivalent, i.e., the user perceives the database as if each object has only one copy. Combined, a concurrency control protocol for non-replicated databases and a replica control protocol ensure one-copy serializability.

A simple concurrency control protocol for replicated databases is the read-one write-all protocol. The first component of this protocol is a two-phase locking protocol [Eswa76] that ensures serializability in non-replicated databases. The replica control component requires a logical write $x$ operation to write all copies of $x$, and a logical read $x$ operation to access any one copy of $x$. This concurrency control protocol ensures one-copy serializability in replicated databases; however, if any object has a copy residing on a failed site, no write operation may be executed on that object.

The available copies protocol [Bern84, Good83] overcomes this problem by requiring a logical write of $x$ to write only copies of $x$ residing on operational sites. Hence, with this protocol write operations can be executed even when sites fail. To ensure one-copy serializability, this protocol requires a special exclude transaction
be executed to inform other sites of the failure of a site. A corresponding *include transaction* is executed whenever a site recovers, to update all the copies residing on the recovered site, and to inform other sites to resume write operations on the recovered copies. This protocol ensures one-copy serializability and tolerates site failures. However, if partitioning failures occur in the network, the available copies protocol may lead to non-serializable executions.

*Gifford's quorum protocol* [Giff79] maintains one-copy serializability and also tolerates site and link failures, even when these failures lead to partitioning. A write of object $x$ is not required to write all copies of $x$, but only a *write quorum*, $q_w[x]$, copies of $x$. A read of $x$ is executed by accessing a *read quorum*, $q_r[x]$, copies of $x$ and reading the most up-to-date copy. Any set of size $q_w[x]$ must have at least one copy in common with any set of size $q_w[x]$ or $q_r[x]$. In this protocol, quorums are fixed and determine both the availability of an object for reading and writing, as well as the cost of executing operations on that object. Furthermore, in order to tolerate failures, write operations on an object should not have to write all the copies of that object, and thus read operations must access more than one copy.

In this thesis we develop and study protocols that are designed to provide the database designer with the flexibility of determining the degree of data availability separately from the costs of operation execution. These protocols ensure one-copy serializability even if network partitioning failures occur.
CHAPTER 3

The Virtual Partitions Protocol

3.1. Introduction

The goal of this chapter is to develop an intuition that will simplify the understanding of the results in Chapter 4. We develop the virtual partitions protocol for managing replicated data in the presence of site, link and partitioning failures. The presentation of the protocol is informal, and the proofs are only outlined. In the next chapter, we formally describe and rigorously prove a more efficient and flexible protocol.

The virtual partitions protocol reduces the cost of read and write operations by using information about the state of the network. Unlike Gifford's quorum protocol [Giff79], in which a logical read operation physically accesses several copies of an object, this protocol translates every logical read operation to a single physical access operation. To achieve this, the protocol adapts dynamically to topological changes in the network. Each site maintains a view of the current communication topology. Views are used to optimize the translation of logical operations into physical operations. Although views should ideally reflect the actual communication topology, instantaneous detection of failures and recoveries is not possible. Hence, a site's view is an approximation of the set of sites with which it can communicate. These views are maintained by a specialized view management protocol.
We start by making some idealized assumptions about views, and how they track changes in the network. We present a simple concurrency control protocol that ensures correctness under these ideal assumptions. We then assume a more realistic model and modify the simple protocol accordingly.

3.2. A Protocol for an Idealistic Network

To simplify the presentation, we introduce the concept of a communication graph. The nodes of the graph represent sites, and an undirected edge between two nodes $s$, $p$ indicates that if $s$ and $p$ send messages to each other, these will be received within a specified time limit. A connected component of a communication graph is called a communication cluster. A partition failure results in a graph containing two or more clusters.

To adapt to changes in the communication topology, each site maintains a local "view": the view of site $s$, $\text{view}(s)$ is the set of sites $s$ assumes it can communicate with.

In order to understand better what is attainable, let us first consider a failure model in which changes in the communication topology (resulting from failures and recoveries) are instantly detected by all affected sites.

(A1) The view of each site contains all the sites adjacent to it in the current communication graph and no other sites.

Thus, from A1, it follows that the views of sites in the same partition are equal and the views of sites in different partitions are disjoint.
Given the above assumption, we now present a simple concurrency control protocol for the execution of transactions in a replicated database. We will assume that transactions use a conflict preserving concurrency control protocol for non-replicated databases, such as two phase locking [Eswa76]. Such a protocol ensures a serialization order on all transactions that issue physically conflicting operations. In a replicated database, however, we have to ensure a serialization order on all transactions that issue logically conflicting operations. We therefore introduce the following rules.

Consider a transaction \( t_i \) that is initiated at site \( s \). To execute a read operation \( r_i[x] \), transaction \( t_i \) first determines whether a majority of copies of \( x \) resides on sites in \( \text{view}(s) \), and if so \( t_i \) accesses one copy of \( x \) residing on a site in \( \text{view}(s) \). To execute a write operation \( w_i[x] \), transaction \( t_i \) first determines whether a majority of copies of \( x \) resides on sites in \( \text{view}(s) \), and if so \( t_i \) writes all copies of \( x \) residing on sites in \( \text{view}(s) \). Transaction \( t_i \) is aborted if any access or write operation cannot be executed. Each transaction uses a commit protocol to ensure its atomic execution. Finally, a recovery transaction is executed in all newly formed partitions. For each object with a majority of copies in the new partition, this transaction accesses all copies, reads the most up-to-date value, and writes that value into all copies residing on sites in the partition.

It is easy to show that the above protocol ensures one-copy serializability. Assumption A1 ensures that all sites in a partition have the same view, which contains all sites in that partition. The "read-one/write-all rule" ensures that, in each partition, any two logically conflicting operations also physically conflict. The
"majority rule" ensures that only transactions executing in one partition can read or write an object at a time. Hence for each object \( x \), any two partitions with transactions that read or write \( x \) must have at least one site in common. Any write \( x \) writes all copies in the partition, and hence must physically conflict with any other write \( x \) (including those executed in other partitions). Furthermore, any read \( x \) either reads a value written by a transaction in the same partition, or a value written by a recovery transaction. But the recovery transaction accesses all copies of an object \( x \) in a partition, and hence must physically conflict with all write \( x \) operations executed in other partitions. Therefore, any two logically conflicting operations must either physically conflict on some copy, or they indirectly conflict through a set of recovery transactions. Since all transactions use a conflict preserving concurrency control protocol, these rules ensure that all transaction executions are one-copy serializable.

The above rules are simple, and ensure a high level of data availability, provided the communication information maintained by the sites is accurate. Unfortunately, the correctness of the rules depends heavily on assumption A1. If it is relaxed, one-copy-serializability cannot be guaranteed.

**Example 1.** Consider a network where assumption A1 no longer holds. Consider three sites \( s_1, s_2 \) and \( s_3 \), where \( s_1 \) assumes it can only communicate with \( s_3 \), \( s_2 \) assumes it can only communicate with \( s_3 \), and \( s_3 \) assumes it can with both \( s_1 \) and \( s_2 \). We thus have: \( \text{view}(s_1) = \{s_1, s_3\} \), \( \text{view}(s_2) = \{s_2, s_3\} \) and \( \text{view}(s_3) = \{s_1, s_2, s_3\} \).

Suppose an object \( x \) has three copies \( x_{s_1}, x_{s_2} \) and \( x_{s_3} \), which are initialized to 0. Each site will consider \( x \) to be accessible, since each has a majority of the copies in
its view. Now, suppose two transactions $t_1$, initiated at $s_1$ and later, $t_2$ initiated at $s_2$, increment $x$ by 1. Based on the view of $s_1$, $t_1$ reads copy $x_{s_1}$ and updates both that copy and $x_{s_3}$. Similarly, $t_2$ reads copy $x_{s_2}$ (which still contains 0) and updates both that copy and $x_{s_3}$. Observe that after two successive increments, all copies of $x$ contain 1. Clearly, the execution of these transactions is not one-copy serializable.

**Example 2.** Consider a network with four sites, $s_1$, $s_2$, $s_3$ and $s_4$, and four objects $w$, $x$, $y$, and $z$. Object $w$ has two copies residing on site $s_1$ and one copy residing on $s_4$, object $x$ has two copies residing on site $s_2$ and one copy residing on $s_1$, object $y$ has two copies residing on site $s_3$ and one copy residing on $s_2$, and object $z$ has two copies residing on site $s_4$ and one copy residing on $s_3$. Assume that the network is partitioned into clusters $\{s_1,s_2\}$ and $\{s_3,s_4\}$, where sites $s_1$ and $s_2$ have view $\{s_1,s_2\}$, and sites $s_3$ and $s_4$ have view $\{s_3,s_4\}$. Assume the network undergoes repartitioning such that $s_2$ and $s_3$ form a partition, and $s_1$ and $s_4$ form another partition. However, only sites $s_2$ and $s_4$ detect the occurrence of the new partition immediately and update their views. Site $s_2$ updates its view to $\{s_2,s_3\}$, and site $s_4$ updates its view to $\{s_1,s_4\}$. The other two sites ($s_1$ and $s_3$) keep views $\{s_1,s_2\}$ and $\{s_3,s_4\}$ respectively. Assume the following transactions are executed: transaction $t_1$, which is initiated at $s_1$, reads $x$ and write $w$, transaction $t_2$, which is initiated at $s_2$, reads $y$ and write $x$, transaction $t_3$, which is initiated at $s_3$, reads $z$ and write $y$, and transaction $t_4$, which is initiated at $s_4$, reads $w$ and write $z$.

Consider the execution of $t_1$ initiated at site $s_1$. Since $s_2$ and $s_1$ are in $\text{view}(s_1)$, then both $w$ and $x$ are accessible in that view, and $t_1$ can read $x_{s_1}$. Since
$s_1$ has two copies of $w$, $t_1$ can update them, and, furthermore, $t_1$ will not attempt to update $x_{s_4}$ since $s_4 \notin \text{view}(s_1)$. Hence, in the execution of transaction $t_1$, $t_1$ accesses and writes only copies that reside on $s_1$. The execution of transactions $t_2$, $t_3$, and $t_4$ proceed similarly, with each transaction accessing only copies residing on the initiating site. Since no communication with remote copies are made, the inconsistency between the sites’ views is not detected. The resulting execution is serializable but not one-copy serializable.

As example 1 illustrates, the correctness of the protocol depends on the property that no two sites with different views are able to access a common set of copies. Example 2 illustrates that even in a communication network, where only sites with the same view communicate, sites can not independently and asynchronously update their views.

### 3.3. The Replica Control Protocol

We now introduce a replica control protocol that uses the majority and the read/write rules mentioned above, and circumvents the anomalies illustrated in examples 1 and 2 by restricting when and how sites may update their views. Toward this goal, we introduce the notion of a virtual partition. Roughly speaking, a *virtual partition* is a set of communicating sites that have agreed on a common view and on a common way to test for membership in the partition. For the purposes of transaction processing, only sites that are assigned to the same virtual partition may communicate. Hence, a virtual partition can be considered a type of "abstract" communication cluster where sites join and depart in a disciplined manner. In contrast, in a real communication cluster sites depart abruptly
because of failures.

The common view of the members in a virtual partition represents a shared estimate of the set of sites with which communication is possible. When a site detects an inconsistency between its view and its communication capabilities (by not receiving an expected message or by receiving a message from a site not in its view), it can unilaterally depart from its current virtual partition. After departing, the site can invoke a protocol to establish a new virtual partition. This protocol, the view management protocol, creates a new virtual partition, assigns a set of sites to the virtual partition, and updates those sites' views. If the view management protocol changes the view of a site $s$ to $v$, we say that $s$ joins the virtual partition whose sites have view $v$. If the view management protocol changes the view of a site $s$ from $v$, we say that $s$ departs from the virtual partition whose sites have view $v$.

In order to ensure that the simple "read-one/write-all" protocol maintains one-copy serializability, we require that the view management protocol ensure the following property:

(S1) Serialization of Virtual Partitions. Virtual partitions are totally ordered according to some total order, $<<$, which satisfies the following condition:

Let $vp_1$ be a virtual partition whose sites have view $v_1$, and $vp_2$ be a virtual partition whose sites have view $v_2$. Let $s$ be a member of virtual partition $vp_1$ and view $v_2$. If $vp_1 << vp_2$ then $s$ must depart from $vp_1$ before any site $p$ can join $vp_2$. 
This property says that before a new partition $vp_2$, whose sites have view $v_2$ can be formed, every site $s$ in $v_2$ must first depart from its current virtual partition. Note that the property does not require $s$ to eventually join $vp_2$. $S_1$ prevents anomalies of the type illustrated in Example 2, where a site $s_2$ detects changes in its communication capabilities, adopts a new view, and begins processing transactions before another site $s_3$ in the same communication cluster detects the changes.

The simple replica control protocol mentioned previously can now be reformulated as follows:

(R1) Majority rule. An object $x$ is accessible from a site $s$ only if a majority of copies of $x$ reside on sites in $view(s)$.

(R2) Read rule. Let $t$ be a transaction that is initiated at site $s$. If an object $x$ is accessible from $s$, then $t$ reads $x$ by accessing any copy of $x$ residing on a site in $view(s)$. If the access operation cannot be executed, it can be retried at another site or the logical read can be aborted.

(R3) Write rule. Let $t$ be a transaction that is initiated at site $s$. If an object $x$ is accessible from $s$, then $t$ writes $x$ by writing all copies of $x$ residing on sites in $view(s)$. If any physical write cannot be executed, the logical write is aborted.

(R4) A transaction $t$, initiated at a site $s$ may only execute physical operations on copies that reside on sites in virtual partition $vp$ whose view is
view(s). In this case we say that \( t \) executes in \( vp \).

Rules R1-R3 are straightforward interpretations of the simple rules presented in the previous section. Rule R4 expresses the communication restriction that is placed on virtual partitions. In particular, a transaction \( t \) initiated at site \( s \) is allowed to execute physical operations on copies that reside on site \( p \) only if sites \( s \) and \( p \) are assigned to the same virtual partition. Rule R4 can be enforced as follows. Each view is assigned a unique \( \text{view-id} \), and when a transaction is initiated at some site \( s \), the transaction is assigned the view-id associated with \( \text{view}(s) \). The transaction includes this view-id with all its physical operations. A site ignores requests with view-ids different from its own.

The final rule concerns updating copies residing on sites that have joined a new virtual partition.

(R5) Partition initialization rule. Let \( p \) be a site that has joined a new virtual partition \( vp \) whose sites have view \( v \). Before any copy \( x_p \) of accessible object \( x \) in \( vp \) can be read or written by a transaction executing in \( vp \), a recovery transaction must be executed that writes into \( x_p \) the most recent value of \( x \) written by any transaction.

Note that the desired value of \( x \) can be found at one of the sites in \( vp \) because a majority of \( x \)'s copies must reside on sites in \( vp \) for \( x \) to be accessible in \( vp \) and a majority of \( x \)'s copies is written by each logical write of \( x \). For each accessible object \( x \), the recovery transaction first accesses all copies of \( x \) residing on sites in the virtual partition. It then determines \( \text{vnmax} \), the highest version number of all
accessed copies and writes the value of the copy with version number \( v_{\text{max}} \) to all copies of \( x \) in the virtual partition. All read and write operations must be delayed between the time that a site joins a new virtual partition and the assignment of the most recent value to its local copies.

We now informally argue the correctness of the protocol presented above. In Chapter 5, we prove that it is an instance of our paradigm, and hence correct (another formal proof is presented in [Abba85]). In the next chapter we present a more efficient and flexible protocol and formally prove its correctness.

We claim that all transaction executions are one-copy serializable. All transactions executing in the same virtual partition are one-copy serializable, and transactions executing in different virtual partition are serialized according to the total order \( \ll \) defined by S1 on the virtual partitions.

Consider transactions executing in the same virtual partition. From R2 and R3, transactions use a read-one write-all copies in the virtual partition. From R4, all copies accessed or written must reside on sites in the same virtual partition. Hence, all logically conflicting operations executed in a virtual partition also conflict physically. Since all transactions use a conflict preserving concurrency control protocol, then all transactions executing in the same virtual partition are one-copy serializable.

Consider transactions executing in different virtual partitions. Let \( t_i \) and \( t_j \) be two logically conflicting transactions that read or write the same object \( x \), and that execute in virtual partitions \( vp_i \) and \( vp_j \), where \( vp_i \ll vp_j \). There are two cases to consider, depending on whether \( t_i \) reads or writes \( x \).
First consider the case where \( t_i \) writes \( x \). From R4, all copies accessed or written by \( t_j \) must reside on sites in virtual partition \( vp_j \). From R5, a recovery transaction \( t_r \) must update each copy residing on a site in \( vp_j \) before any user transaction can execute in \( vp_j \). Hence, there must be at least one copy that is written by \( t_r \) before being accessed or written by \( t_j \), and therefore \( t_r \) must be serialized before \( t_j \). We now show that \( t_r \) must be serialized after \( t_i \), and hence, \( t_j \) must be serialized after \( t_i \). Recovery transaction \( t_r \) accesses at least a majority of copies of \( x \), and user transaction \( t_i \) writes at least a majority of copies of \( x \). Hence, there must be at least one copy \( x_s \) that \( t_r \) accesses and \( t_i \) writes. From R4, \( s \) must be a member of \( vp_i \) when \( t_i \) is executed. From R5, \( s \) must be a member of \( vp_j \) when \( t_r \) is executed. From S1, since \( vp_i \ll vp_j \), \( s \) cannot be a member of \( vp_i \) once \( vp_j \) has been created. Hence, \( t_i \) must have written \( x_s \) before \( t_r \) accesses \( x_s \), and therefore, \( t_r \) must be serialized after \( t_i \).

Second, consider the case where \( t_i \) reads \( x \). Assume that \( t_i \) reads a value of \( x \) written by some transaction \( t_k \). Since \( t_i \) and \( t_j \) logically conflict and \( t_i \) reads \( x \), then \( t_j \) must write \( x \). Both \( t_j \) and \( t_k \) write a majority of copies of \( x \), hence there must be at least one copy that they both write. But \( t_i \) reads a value of \( x \) written by \( t_k \), hence, \( t_k \) must have executed in a virtual partition that precedes \( vp_i \) in the total order \( \ll \), and therefore \( vp_k \ll vp_i \). But \( vp_i \ll vp_j \), hence, \( vp_k \ll vp_j \). As in the previous case, \( t_k \) must be serialized before \( t_j \), therefore \( t_i \) reads a value of \( x \) that was changed by \( t_j \) and hence, \( t_i \) must be serialized before \( t_j \).

Note that since views do not instantaneously track changes in the network, it is possible for an object \( x \) to be accessible in two virtual partitions at the same
time. This can occur because views of different virtual partitions that exist concurrently can overlap, and hence, the same object can be accessible in these partitions. However, at most one of these partitions will be able to write the object, since a logical write operation writes a majority of the copies that reside on sites in the same virtual partition, and no two partitions can concurrently have a majority of copies. A logical read is required only to access one copy, hence many partitions may be able to read the same object and, therefore, will be able to read out-of-date values. Hence, although the virtual partitions protocol produces only one-copy serializable executions, some transactions may not read the most up-to-date (in real time) copies of objects. This phenomenon is not detectable by applications executing transactions since, by design, applications can not send messages across partition boundaries. This might, however, be detected by a user that moves from a site in one partition to a site in another [Fisc82, Weih85].

The possibility that a site will read data, which is not up-to-date, can arise when a site's view no longer accurately reflects the network topology. This situation arises most often when a site is slow to detect the occurrence of a failure. For most systems, such situations can be expected to be short-lived, and if desired, a distributed systems can periodically send probe messages in order to bound this phenomenon [Cris84].

3.4. The View Management Protocol

We now describe a view management protocol that satisfies S1. A key aspect of the protocol is the implementation and generation of view-ids. A view-id consists of two fields \( \langle no, id \rangle \) where \( no \) is an integer greater than any other \( no \) the
initiator of the new view has encountered, and id is the unique site identifier of the initiator. We say <no_1, id_1> is less than <no_2, id_2>, if no_1 < no_2, or no_1 = no_2 and id_1 < id_2. The view-ids define a total order on all views, and hence, on virtual partitions. Our protocol will ensure that this total order satisfies S1.

The view management protocol, schematically illustrated in Figure 3.1, requires two phases. In the first phase, some site, which we will call the initiator, initiates a new virtual partition by generating a new view-id, v_id, for the new view that it will attempt to establish. It then sends an invitation to join the new view to all sites. A site accepts the invitation only if it has not already received another invitation to join a virtual partition whose view has a higher view-id.

---

**Initiator**

\[ v_{id} := \text{generate-new-view-id()} \]

for each site \( p \)

send("invite",v_id) to \( p \)

receive("invite",v_id) from initiator

if \( v_{id} > \max\text{imum-view-id-ever-seen} \)

then depart current virtual partition

reply("accept",v_id)

\[ \text{new\_view} := \{ p : p \text{ replies "accept"} \} \]

for each \( p \) replying "accept"

send("join",v_id,new\_view)

receive("join",v_id,new\_view) from initiator

if \( v_{id} > \max\text{imum-view-id-ever-seen} \)

then join \( v_{id} \) and adopt new\_view

---

Figure 3.1. The View Management Protocol
After accepting or initiating an invitation and before committing itself to a new virtual partition, a site is not assigned to any virtual partition.

In the second phase, the initiator defines the new view to be the set of sites accepting the invitation. Let us denote this set by $A$. The initiator sends the new view to all sites in $A$. Upon receiving the new view, $A$, an accepting site assigns itself to the virtual partition with view $A$, and view-id $v_{id}$, only if it has not, in the meantime, accepted an invitation to join a virtual partition whose view has a higher view-id. Hence, at the end of the second phase, only a subset of the sites in $A$ might actually be assigned to the virtual partition with view $A$ and view-id $v_{id}$. (Another reason is that "join" messages from the initiator may be lost.)

Notice that in both phases, a site checks to make sure that the view-id in the message is larger than the maximum view-id seen so far. These checks serve two purposes: they prevent a site from being confused by an old, out-of-date message, and they prevent several sites in the same cluster to simultaneously attempt to establish new virtual partitions. In the absence of additional failures, only the site generating the highest numbered view-id will succeed.

The protocol ensures that the view-ids associated with the views of each virtual partition define a total order $\ll$ on all virtual partitions. This total order satisfies the conditions of S1: a site $s$ does not install a new view containing a site $p$, until it has received an acknowledgment from $p$. But a site $p$ that accepts an invitation departs from its current virtual partition, and stops using its view. Furthermore, the view-id associated with the new view is greater than any view-id associated with any previous view of a site in the new view. Hence, the total order
defined by the view-ids satisfies condition S1.

3.5. Conclusion

We have presented a simple fault-tolerant concurrency control protocol that ensures one-copy serializability. The protocol tolerates site and link failures that might lead to partitioning. The implementation of reads is efficient even in the presence of failures. The view management protocol makes these failures look like "clean" communication failures that partition the network. This allows us to introduce the notion of virtual partitions and to base our protocol on the rules that (1) a transaction initiated at a site in a virtual partition can read or write an object \( x \) if a majority of copies of \( x \) reside on sites in the virtual partition, and (2) transactions use the simple "read-one write-all" rule to translate logical operations. In the next chapter, we further develop the ideas presented in this chapter and propose a more efficient protocol.
CHAPTER 4

The Accessibility Thresholds Protocol

4.1. Introduction

In this chapter, we present a new concurrency control protocol for replicated objects that allows the accessing of data even when the database is partitioned. The replica control component of our protocol can be combined with any conflict preserving concurrency control protocol for non-replicated objects to ensure the correctness of a database. Thus, the replica control protocol we propose is compatible with any of the protocols used in distributed systems such as $R^*$ [Lind84], distributed INGRES [Ston86], DDM [Good84], or any database that uses locking protocols. In contrast to the virtual partitions protocol presented in the previous chapter, this protocol does not require a separate sub-protocol to coordinate the views different sites have of the communication network. This results in a simpler and more efficient protocol. Furthermore, we show that this protocol provides a higher degree of data availability, and flexibility in determining operation costs.

With this protocol, as with the virtual partitions protocol, it is never necessary for a read operation to access more than one copy, even if the database partitions. This is of vital importance in any practical system where read operations outnumber write operations, and hence efficient read operations are critical for the success of the implementation. The cost of a read operation is independent of the level of availability associated with read or write operations. In general, this pro-
tocol provides the database designer with a larger degree of flexibility in deciding when operations may be executed on objects, as well as in deciding the costs of these operations. It also provides a higher degree of data availability than in [Giff79, Eage83].

In the next section, we describe the proposed protocol, and in Section 4.3 we prove it correct. Section 4.4 presents several optimizations to the protocol. A comparison with other concurrency control protocols and a discussion concludes the chapter.

4.2. The Replica Control Protocol

Our replica control protocol assumes two types of transactions: user transactions, issued by the users of the database, and update transactions, issued by the protocol. We assume that all transactions follow a conflict preserving concurrency control protocol for non-replicated databases, e.g., two-phase locking [Eswa76]. Such a protocol ensures logs are CP-serializable. In this section we present a replica control protocol that ensures that all logs are one-copy serializable.

To describe the execution of transactions, we introduce the notion of a view [Abba85]. Each site $s$ maintains a set of sites called its view. Each site can independently decide which sites to include in its current view. For example, a site $s$ may include in its view all sites with which it assumes it can communicate. The correctness of our protocol does not depend on this choice, although data availability and operation costs may be affected (in Section 4.2.3, we describe several possible strategies that sites can use to choose their current view, and the tradeoffs involved). A user transaction $t$ that is initiated at a site with view $v$ is said to
execute in \( v \). Informally, the view in which transaction \( t \) executes determines which objects \( t \) can read and write, as well as which copies it can access or write. Views are totally ordered according to a unique \( view-id \), which is assign to each view by the site that determines which sites to include in that view. Two sites are said to have the same \( view \) if they have identical view-ids.

When combined with a conflict preserving concurrency control protocol, our replica control protocol maintains one-copy serializability by ensuring two conditions. First, all transactions executed in one view are one-copy serializable, and second, all transactions executing in a "lower" view are serialized before transactions executing in a "higher" view.

### 4.2.1. Accessibility Thresholds, Quorums and User Transactions

As in [Abba85, Abba86], we associate with each object \( x \), a read and a write accessibility threshold, \( A_r[x] \) and \( A_w[x] \) respectively. An object \( x \) is read (write) accessible in a view only if \( A_r[x] \) (\( A_w[x] \)) copies reside on sites in that view. The accessibility thresholds \( A_r[x] \) and \( A_w[x] \) must satisfy:

\[
A_r[x] + A_w[x] > n[x]
\]  \hspace{1cm} (1)

This relationship ensures that a set of copies of \( x \) of size \( A_w[x] \) has at least one copy in common with any set of copies of \( x \) of size \( A_r[x] \). (In [Abba85, Abba86], the write accessibility thresholds have to satisfy the additional requirement that \( 2A_w[x] > n[x] \), and this requirement prevents transactions in more than one partition from writing \( x \) concurrently.)

In each view \( v \), every object \( x \) is assigned a read and write quorum, \( q_r[x,v] \) and \( q_w[x,v] \): these specify how many physical access and write operations are needed to
read and write object $x$ in view $v$. Let $n[x,v]$ be the number of copies of $x$ that reside on sites in view $v$; formally, $n[x,v] = |\text{sites}[x] \cap v|$. For each view $v$, the quorums of object $x$ must satisfy the following relations:

$$q_r[x,v] + q_w[x,v] > n[x,v]$$  \hspace{1cm} (2)

$$1 \leq q_r[x,v] \leq n[x,v]$$ \hspace{1cm} (3)

$$2q_w[x,v] > n[x,v]$$ \hspace{1cm} (4)

$$A_w[x] \leq q_w[x,v] \leq n[x,v]$$ \hspace{1cm} (5)

These relations ensure that, in a view $v$, a set of copies of $x$ of size $q_w[x,v]$ has at least one copy in common with any set of copies of $x$ of size $q_r[x,v]$, $q_w[x,v]$, and $A_r[x]$.

Read operations use the version numbers associated with each copy to identify (and read) the most "up-to-date" copy accessed (as defined in Section 2.2). In our protocol, version numbers consist of two fields $<v\_id,k>$ . Intuitively, if a copy has version number $<v\_id,k>$, then this copy was last written by a transaction $t$ executing in a view $v$ with view-id $v\_id$, and $t$ is the $k^{th}$ transaction to write $x$ in view $v$. A version number $<v_1\_id, k_1>$ is less than $<v_2\_id, k_2>$, if $v_1\_id < v_2\_id$, or $v_1\_id = v_2\_id$ and $k_1 < k_2$. Initially, sites have a common view $v_0$ with view-id $v_0\_id$, and all copies have version number $<v_0\_id,0>$. We now describe how user transactions execute read and write operations according to our protocol.

A user transaction $t$ executing in view $v$ can read (write) an object $x$ only if $x$ is read (write) accessible in view $v$. Furthermore, $t$ can only access or write copies of $x$ that reside on sites with view $v$ (this restriction will be relaxed in Section
4.4.1. If object $x$ is read accessible in view $v$, $t$ executes the logical operation $r[x]$ by:

1. physically accessing $q_r[x,v]$ copies of $x$ residing on sites in $v$ (with view $v$),
2. determining $vnmax$, the maximum version number of the selected copies, and
3. reading the accessed copy with version number $vnmax$.

If object $x$ is write accessible in view $v$, with view-id $v\_id$, $t$ executes the logical operation $w[x]$ by:

1. selecting $q_w[x,v]$ copies of $x$ residing on sites in $v$ (with view $v$),
2. determining $vnmax$, the maximum version number of the selected copies, and
3. writing all the selected copies and updating their version numbers to $<v\_id,l>$, where $l \geq 1$ is the smallest integer such that $<v\_id,l>$ is greater than $vnmax$.

Quorum relations (2) and (4) ensure that all logically conflicting operations issued by user transactions executing in the same view, also conflict physically. Furthermore, since all transactions use version numbers and a conflict preserving concurrency control protocol that would work for a non-replicated database, one can show that all transactions executing in the same view are one-copy serializable. Note that all transactions use a commit protocol to ensure atomicity, and as we noted earlier there are no non-blocking protocols that tolerate partitioning failures [Skee82], hence a transaction may not terminate if partitioning failures occur during its execution. Several methods have been develop to reduce the possi-
bility of blocking [Lind84, Cheu86], which could be use with our protocol. Furthermore, since we assume that transactions may use any conflict preserving concurrency control protocol for non-replicated database, the protocol developed by Joseph and Birman [Jose85, Jose86] for asynchronous update may be use to execute write operations.

The use of accessibility thresholds in conjunction with quorums, and the fact that each view can independently define its own quorums for each object gives the database designer an unusual degree of flexibility, as we first observed in [Abba86]. This can be used to achieve the desired cost/availability tradeoff of read and write operations. There are several such tradeoffs. For example, one can increase the availability of an object for reading by decreasing the read accessibility threshold, $A_r[x]$—at the cost of increasing $A_w[x]$, i.e., decreasing the availability of write operations on the object. In some applications read operations on some object $x$ outnumber write operations, and in this case it is advantageous to allow inexpensive read operations for $x$. Using quorums, this can be easily achieved with $q_r[x,v] = 1$, and $q_w[x,v] = n[x,v]$. A more detailed discussion of the possible tradeoffs in choosing the accessibility thresholds and quorums will be presented in Sections 4.2.3, 4.4.1.2 and 4.4.2.

4.2.2. Update Transactions

Views change during system execution. For example, a site may want to change its view when it notices a discrepancy between its current view and the sites it can actually communicate with (again, this is not necessary for correctness, but may improve performance). A site $s$ changes its view in one of two ways. Site
s may decide on the members of a new view v based on its own information, in
which case s is called the initiator of v. (Policies for determining when to initialize
a new view, and which sites to include in the new view will be discussed in the
next section.) Site s may also decide to adopt a view v initiated by another site, in
which case we say s inherits view v. Whenever a site s changes its view to a new
view v, either by initiating that view or by inheriting it, s must execute an update
transaction that updates the local copies residing on site s. Views are considered
objects, therefore a transaction that executes an operation on a view must follow
the conflict preserving concurrency control protocol.

Informally, sites change their views as follows. When a site s decides to ini-
tiate a new view, it first assigns to the new view a unique view-id new_view_id
that is larger than any other the initiator has encountered (uniqueness can be
achieved by appending the initiator's site-identifier to the view-id). Site s then
executes an update transaction to update all its local copies. For each object x that
is read accessible in the new view, this update transaction accesses enough copies
of x to have at least one copy in common with the copies written by write opera-
tions initiated in previous views (sites with a view-id greater than new_view_id
reject these access operations). The update transaction reads the value of the
accessed copy of x with the highest version number, and writes it to the local copy
x_s. If the update transaction is terminated successfully then s installs the new
view, and user transactions may execute in the new view and access or write copies
residing on s. All sites accessed by the update transaction either have the same
new view or they immediately try to inherit this view by executing an update
transaction. We now present in more detail the process of changing views.

Let $s$ be a site whose current view has view-id $\text{current\_view\_id}$. To initiate a new view, $s$ atomically executes the procedure $\text{initiate\_new\_view}(\text{current\_view\_id})$ illustrated in Figure 4.1. Site $s$ chooses a new view, $\text{new\_view}$, and determines an associated view-id, $\text{new\_view\_id}$, higher than the current view-id. Then, $s$ tries to install this new view by executing $\text{install\_view}(\text{new\_view}, \text{new\_view\_id})$. If it fails, it initiates a new view with a higher view-id by calling $\text{initiate\_new\_view}$ recursively.

Procedure $\text{install\_view}$ (see Figure 4.2) executes an update transaction to update local copies of objects. If the update transaction is successful, $s$ installs $\text{new\_view}$ by updating $\text{current\_view}$ and $\text{current\_view\_id}$ to $\text{new\_view}$ and $\text{new\_view\_id}$, respectively. If the update transaction fails $\text{install\_view}$ is aborted. Once $\text{new\_view}$ is installed at a site $s$, user transactions initiated at $s$ are allowed to execute in $\text{new\_view}$.

```c
/* initiates and installs a new view at site s */
/* with view-id greater than current_view_id */
initiate_new_view(current_view_id)

new_view := \{set of sites\}
new_view_id := increment (current_view_id)
if install_view(new_view, new_view_id) is aborted
   \rightarrow initiate_new_view(new_view_id)
fi
```

**Figure 4.1. initiate_new_view procedure**
/* tries to install new_view at site s with new_view_id */
install_view(new_view, new_view_id)

execute update transaction(new_view, new_view_id)
if update transaction is not aborted /* install new_view */
   \[ current_view := new_view \]
   \[ current_view_id := new_view_id \]
[] update transaction is aborted
   \[ abort install_view \]
fi

Figure 4.2. install_view procedure

/* tries to update the local copies residing on site s */
/* of all read accessible objects in new_view */
update_transaction(new_view, new_view_id)

for all read accessible objects x in new_view do
   select a set of \( A_s[x] \) copies of x including \( x_s \)
   for all selected copies \( x_p \) execute access(\( x_p \), new_view, new_view_id)
   if no access operation aborted
      \[ \text{read accessed copy of } x \text{ with highest version number } v_{\text{max}} \]
      write \( x_s \) with the value read and version number
      \[ <\text{new_view_id}, l> \text{ where } l \geq 0 \text{ is the smallest integer such that} \]
      \[ <\text{new_view_id}, l> \geq v_{\text{max}} \]
[] some access operation aborted
   \[ abort update transaction \]
fi
od

Figure 4.3. update transaction
Update transactions ensure that local copies of read accessible objects are up-to-date (see Figure 4.3). \texttt{Update\_transaction(new\_view, new\_view\_id)} executed at site \(s\) updates all the local copies of objects that are read accessible in \textit{new\_view}. For each object \(x\) that is read accessible in \textit{new\_view}, it first accesses \(A_{s}[x]\) copies of \(x\) (including \(x_{s}\)) and determines \(vnmax\), the maximum version number of the selected copies. It then reads the copy associated with \(vnmax\) and writes it into \(x_{s}\) with version number \(<\textit{new\_view\_id},l>\), where \(l \geq 0\) is the smallest integer that is greater than or equal to \(vnmax\). Since there are no non-blocking protocols that always guarantee transaction termination when partitioning failures occur, the update transaction may block in the rare event that such failures occur during its execution.

We associate with each access operation the new view, \textit{new\_view}, and its view-id, \textit{new\_view\_id}. The access of copy \(x_{p}\) is aborted if \(s\) cannot communicate with \(p\), or if \(p\) has a view whose view-id is higher than \textit{new\_view\_id}. If any access operation aborts, the update transaction is also aborted.

When a site \(p\) receives a request from site \(s\) to access \(x_{p}\), \(p\) can take three possible actions depending on \textit{new\_view\_id}, the view-id associated with the request, and \textit{current\_view\_id}, the view-id associated with \(p\)'s current view (see Figure 4.4 where each branch of the if statement is atomically executed). If \textit{current\_view\_id} is less than \textit{new\_view\_id} then \(p\) executes the access operation, and immediately executes procedure \textit{inherit\_view}. No other user transaction can access or write \(x_{p}\) before \(p\) terminates the \textit{inherit} procedure. If \textit{current\_view\_id} is equal to \textit{new\_view\_id} then \(p\) executes the access operation. If \textit{current\_view\_id} is greater
/*Response of site \( p \) with view \( \text{current}_\text{view} \) and view-id \( \text{current}_\text{view}_\text{id} \) */
/* to a request \( \text{access} \) by site \( s \) */
\( \text{access}(x_p, \text{new}_\text{view}, \text{new}_\text{view}_\text{id}) \)

\textbf{if} \( \text{current}_\text{view}_\text{id} < \text{new}_\text{view}_\text{id} \)
\hspace*{1em} \textbf{then} \textbf{execute} \( a[x_p] \) and return \( x_p \) to \( s \)
\hspace*{1em} \textbf{if} \( p \neq s \) \textbf{then} \textbf{execute} \( \text{inherit}_\text{view}(\text{new}_\text{view}, \text{new}_\text{view}_\text{id}) \)
\hspace*{1em} \textbf{end} \textbf{if}
\hspace*{1em} \( \text{current}_\text{view}_\text{id} = \text{new}_\text{view}_\text{id} \)
\hspace*{1em} \textbf{then} \textbf{execute} \( a[x_p] \) and return \( x_p \) to \( s \)
\hspace*{1em} \textbf{end} \textbf{if}
\hspace*{1em} \( \text{current}_\text{view}_\text{id} > \text{new}_\text{view}_\text{id} \)
\hspace*{1em} \textbf{then} \textbf{abort} \( \text{access} \) operation
\hspace*{1em} \textbf{end} \textbf{if}

\textbf{fi}

\textbf{Figure 4.4.} Response to an \textit{access} request

than \( \text{new}_\text{view}_\text{id} \) then \( p \) aborts the access operation.

To inherit a view \( \text{new}_\text{view} \) with view-id \( \text{new}_\text{view}_\text{id} \), site \( s \) atomically executes the procedure \( \text{inherit}_\text{view}(\text{new}_\text{view}, \text{new}_\text{view}_\text{id}) \) illustrated in Figure 4.5.

In this procedure, site \( s \) tries to install the new view. If \( \text{inherit}_\text{view} \) is aborted, then \( s \) initiates a view with a higher view-id. Note that when a site initiates or

/* inherits and installs \textit{view} or initiates a new view with a higher view-id */
\( \text{inherit}_\text{view}(\text{view}, \text{view}_\text{id}) \)

\textbf{if} \( \text{install}_\text{view}(\text{view}, \text{view}_\text{id}) \) is aborted
\hspace*{1em} \textbf{then} \textbf{initiate}_\text{new}_\text{view}(\text{view}_\text{id})
\hspace*{1em} \textbf{end} \textbf{if}

\textbf{Figure 4.5.} \( \text{inherit}_\text{view} \) procedure
inherits a new view, the new view-id is always greater than any previous view-ids at that site.

4.2.3. Policies for Changing Views

The protocol described in Figures 4.1, 4.2, 4.3, 4.4 and 4.5 includes only the basic steps necessary for ensuring one-copy serializability. It does not include possible optimizations or options for implementation. In the next section, we prove that this basic protocol ensures one-copy serializability, and then in Section 5, we present several optimizations that can be applied. As we mentioned before, perfect knowledge of a site’s communication capabilities is not necessary for determining which sites are in its view. Furthermore, the protocol’s correctness depends neither on when a view is changed, nor on which sites it includes in a new view. In this section we present an example that illustrates some of the options for changing views, and the tradeoffs involved. We then discuss some possible tracking policies: these determine when a site should change its view, and which sites to include in the new view.

Consider a database with four sites $s_1$, $s_2$, $s_3$ and $s_4$, where object $x$ has three copies residing on sites $s_1$, $s_2$ and $s_3$, and object $y$ has three copies residing on sites $s_2$, $s_3$ and $s_4$. Assume that all accessibility thresholds are equal to 2, i.e., $A_r[x] = A_w[x] = A_r[y] = A_w[y] = 2$. Initially all sites have the same view $v_0 = \{s_1, s_2, s_3, s_4\}$ with read quorums $q_r[x,v_0] = q_r[y,v_0] = 1$ and write quorums $q_w[x,v_0] = q_w[y,v_0] = 3$.

Say that the network now partitions into $P_1 = \{s_1,s_2\}$, and $P_2 = \{s_3,s_4\}$. Sites $s_1$ and $s_2$ in partition $P_1$ have two options. If they install a new view
\{s_1, s_2\} (with read quorum \(q_r[x, v_1] = 1\), and \(q_w[x, v_1] = 2\)) to reflect the partitioning of the network, then \(x\) will remain read and write accessible, but since \(A_r[y] = A_w[y] = 2\), \(y\) will not be read or write accessible in this new view. If \(s_1\) and \(s_2\) keep their old view \(\{s_1, s_2, s_3, s_4\}\), (even though they cannot communicate with \(s_3\) and \(s_4\)), then transactions initiated in \(P_1\) can still read both \(x\) and \(y\). Note that even though \(x\) and \(y\) are also write accessible in \(v_0\) (by definition), no transaction initiated in partition \(P_1\) will be able to write \(x\) or \(y\) (i.e., physically write three copies of those objects) since \(q_w[x, v_0] = q_w[y, v_0] = 3\). In summary, if sites in \(P_1\) install the new view, they will retain the ability to read and write \(x\), but will lose the ability to read or write \(y\). However, if they keep their old view despite their new communication capabilities, they will retain the ability to read \(x\) and \(y\), but any attempt to write \(x\) will fail to achieve the necessary quorum.

Hence, the database designer has several tracking strategies to choose from. Views could track changes in the network topology as closely as possible, thus reducing the risk of aborting transactions that read or write objects that are accessible in the current view. This strategy was called aggressive tracking in [Coan86]. (In our example above, this strategy would lead sites \(s_1\) and \(s_2\) to install the new view \(v_1\).) Another approach is to change a view only if some high priority objects are read accessible in the new view. A variation of this strategy is called lazy tracking in [Coan86]. Our protocol with lazy tracking ensures optimal availability according to an availability measure proposed in [Coan86]. A third approach is to change a view only when some high priority transactions can no longer execute in the old view, but would be able to execute in the new view. We call this approach
demand tracking. Thus, our protocol accommodates several strategies for changing views, and the database designer may dynamically choose the strategy used according to the immediate objectives and needs of the specific database being considered.

4.3. Proof of Correctness

In this section we prove that the correctness of the basic replica protocol described in the previous section. We prove that when it is used with a conflict preserving concurrency control protocol for non-replicated databases, all transaction executions are one-copy serializable. Before proceeding to prove the correctness of the protocol, we extend the standard serialization theory to include both user and update transactions.

4.3.1. Extensions to the Standard Serializability Theory

The standard serializability theory presented in Chapter 2 assumed that only user transactions were executed in the system. In this section, we extend the theory to include both update and user transactions. We redefine reads_x_from relations, serialization graphs and one-copy serialization graphs.

We first extend the reads_x_from relation to include update transactions. Let $L$ be a log over a set of transactions $T$. For any two (user or update) transactions $t_i$ and $t_j$ and object $x$, $t_j$ directly reads_x_from $t_i$ in $L$ if there is a copy $x_p$ such that:

1. $w_i[x_p]$ and $r_j[x_p]$ are in $L$.
2. $w_i[x_p] <_L r_j[x_p]$.
3. There is no $w_k[x_p]$ such that $w_i[x_p] <_L w_k[x_p] <_L r_j[x_p]$.
Let $t_u$ be an update transaction, executed by site $s$, that updates the values of a copy $x_s$. If $t_u$ directly reads $x$ from $t_i$, then $t_u$ reads the value of $x$ written by $t_i$ and writes it into $x_s$. Now, if $t_j$ directly reads $x$ from $t_u$ then $t_j$ reads the value of $x$ written by $t_u$. Since this value was originally written by $t_i$, $t_j$ indirectly reads the value written by $t_i$.

We formalize this concept by extending the definition of reads-from. Let $t_{u_1}, t_{u_2}, \ldots, t_{u_n}$ be a sequence of update transactions and $t_i$ and $t_j$ be two user transactions. We say $t_j$ indirectly reads $x$ from $t_i$ if $t_{u_1}$ directly reads $x$ from $t_i$, $t_{u_2}$ directly reads $x$ from $t_{u_1}$, \ldots, and $t_j$ directly reads $x$ from $t_{u_n}$. We henceforth refer to both directly and indirectly reads $x$ from relations simply as reads $x$ from.

The serialization graph $SG[L]$ for a log $L$ is a directed graph whose nodes are all user and update transactions. $SG[L]$ has an edge between any two user transactions that issue conflicting physical operations. We redefine $1-SG[L]$ to ensure that it has a path between any two user transactions issuing logically conflicting operations. $1-SG[L]$ must have enough edges so that:

1. For each object $x$, $1-SG[L]$ embodies a total order $\Rightarrow_x$ on all user transactions that write $x$.

2. For any two user transactions $t_i$ and $t_j$, if $t_j$ reads $x$ from $t_i$ (directly or indirectly), then $1-SG[L]$ has a path from $t_i$ to $t_j$.

3. For any three user transactions $t_i$, $t_j$ and $t_k$, if $t_j$ reads $x$ from $t_i$ (directly or indirectly) and $t_i \Rightarrow_x t_k$, then $1-SG[L]$ has a path from $t_j$ to $t_k$. 
The proof of Theorem 2 in [Bern83] still hold with the extensions we have made: if the graph $1-SG[L]$ is acyclic, the log $L$ is one-copy serializable.

4.3.2. The Proof

Given a log $L$ of transactions executed using our concurrency control protocol, we first show how to construct a corresponding graph, $1-SG[L]$. Then we show that $1-SG[L]$ is acyclic, and hence $L$ is one-copy serializable.

4.3.2.1. Construction of $1-SG[L]$

Let $L$ be a log over a set of user and update transactions executed using our replica control protocol in conjunction with a protocol ensuring CP-serializability in a non-replicated database. Let $SG[L]$ be the corresponding serialization graph. Note that CP-serializability ensures the acyclicity of $SG[L]$ [Eswa76, Hadz84]. We now show how to extend $SG[L]$ into a $1-SG[L]$ by adding enough edges to satisfy the three requirements described in Section 4.3.1. The next lemma proves that $SG[L]$ already satisfies the second of these three requirements.

**Lemma 4.1:** For any two user transactions $t_i$ and $t_j$ if $t_j$ reads $x$ from $t_i$ then $SG[L]$ has a path from $t_i$ to $t_j$.

**Proof:** If $t_j$ directly reads $x$ from $t_i$, then (by definition) there is a copy $x_s$ that $t_i$ writes and is subsequently read by $t_j$. Hence, $SG[L]$ has an edge from $t_i$ to $t_j$. If $t_j$ indirectly reads $x$ from $t_i$, then there is a sequence of update transactions $t_{u_1}, t_{u_2}, \cdots, t_{u_n}$ such that $t_{u_1}$ directly reads $x$ from $t_i$, $t_{u_2}$ directly reads $x$ from $t_{u_1}, \cdots, t_{u_{n-1}}$ directly reads $x$ from $t_{u_n}$. Therefore $SG[L]$ has an edge from $t_i$ to $t_{u_1}$, from $t_{u_1}$ to $t_{u_2}, \cdots$, and from $t_{u_n}$ to $t_j$.  


We now define an order \( \Rightarrow_x \) on all transactions that write \( x \) as follows:

\[ t_i \Rightarrow_x t_j \text{ where } t_i \text{ and } t_j \text{ write } x, \text{ if and only if the version number } t_i \text{ assigns to } x \text{ is less than the one } t_j \text{ assigns to } x. \]

To show that \( \Rightarrow_x \) defines a total order on all user transactions that write \( x \) we need the following technical lemma.

**Lemma 4.2:** The version number of a copy never decreases.

**Proof:** The version number of a copy \( x_s \) is changed only by a user transaction or an update transaction. In the first case, our write rule for user transactions ensures that the version number of \( x_s \) increases. In the second case, the update transaction ensures that the version number of \( x_s \) does not decrease. \( \square \)

**Lemma 4.3:** If \( t_i \) and \( t_j \) are distinct user transactions that write \( x \), then \( t_i \) assigns to \( x \) a different version number than \( t_j \) (and thus, either \( t_i \Rightarrow_x t_j \) or \( t_j \Rightarrow_x t_i \)).

**Proof:** There are two cases to consider. If \( t_i \) and \( t_j \) execute in the same view \( v \), then both \( t_i \) and \( t_j \) write \( q_w(x,v) \) copies of \( x \) in \( v \). Since \( 2q_w(x,v) > n(x,v) \), there must be at least one copy \( x_s \) in \( v \) that both \( t_i \) and \( t_j \) write. Without loss of generality assume \( t_i \) writes \( x_s \) before \( t_j \) does. By Lemma 4.2, and our write rule, \( t_j \) must write \( x \) with a higher version numbers than \( t_i \). If \( t_i \) and \( t_j \) execute in different views, then by definition the first field of the version numbers they assign to \( x \) must be different. \( \square \)

Lemma 4.1 shows that \( SG[L] \) already satisfies the second requirement of a \( 1 - SG[L] \). We now prove that \( SG[L] \) partially satisfies the first requirement as well.
Lemma 4.4: If $t_i \Rightarrow_x t_j$, and $t_i$ and $t_j$ are two user transactions executing in the same view, then $SG[L]$ has an edge from $t_i$ to $t_j$.

Proof: Suppose $t_i$ and $t_j$ execute in view $v$. Thus, both $t_i$ and $t_j$ write $q_w[x,v]$ copies of $x$ in $v$. Since $2q_w[x,v] > \nu[x,v]$, there must be at least one copy $x_s$ in $v$ that both $t_i$ and $t_j$ write. Since $t_i \Rightarrow_x t_j$, $t_i$ writes $x_s$ with a smaller version number than $t_j$. Thus, from our rules for write operations, it is clear that $t_i$ writes $x_s$ before $t_j$ does. Hence, $SG[L]$ contains an edge from $t_i$ to $t_j$. \qed

To prove that $SG[L]$ partially satisfies the third requirement of an $1-SG[L]$, we first need the following technical lemma. To simplify the presentation, we define $v[t]$ as follows. If $t$ is a user transaction, then $v[t]$ is the view $t$ executed in. If $t$ is an update transaction, then $v[t]$ is the view whose installation caused the execution of $t$. Let $v_id[t]$ be the view-id of $v[t]$.

Lemma 4.5: If an update transaction $t_u$ reads $x$ from $t_i$ and $t_i \Rightarrow_x t_k$, where $t_k$ is a user transaction, then $t_u \Rightarrow_x t_k$.

Proof: There are two cases to consider:

(1) Suppose that $t_u$ directly reads $x$ from $t_i$. We first show that there is a copy $x_s$ that is first accessed by $t_u$ and later written by $t_k$. Update transaction $t_u$ accesses $A_r[x]$ copies of $x$ and user transaction $t_k$ writes $q_w[x,v] \geq A_w[x]$ copies of $x$ (for some $v$). Since $A_r[x] + A_w[x] > \nu[x]$ (the total number of copies of $x$), there must be at least one common copy $x_s$ that $t_u$ accesses and $t_k$ writes. Since $t_i \Rightarrow_x t_k$ then, by the definition of $\Rightarrow_x$, the copies of $x$ that $t_i$ writes have a smaller version number than the copies written by $t_k$. By Lemma 4.2, the version number of a copy never decreases. Since $t_u$ directly reads $x$ from $t_i$ and not from $t_k$, it must be
that \( t_u \) accesses \( x_s \) before \( t_k \) writes it.

We claim that \( t_k \) assigns a greater version number to \( x \) than \( t_u \) does (hence, \( t_u \Rightarrow_x t_k \)). Since \( t_u \) directly reads the value of \( x \) written by \( t_i \), then \( t_u \) reads a copy with version number \( v_{\text{id}}[t_i] = \langle v_{\text{id}}[t_i], k \rangle \), where \( v_{\text{id}}[t_i] \leq v_{\text{id}}[t_u] \) and \( k \geq 0 \). Suppose \( v_{\text{id}}[t_u] = v_{\text{id}}[t_i] \). In this case, the update transaction \( t_u \) writes \( x \) with version number \( <v_{\text{id}}[t_i], k > \), the same version number that \( t_i \) assigned \( x \). Thus, since \( t_i \Rightarrow_x t_k \), we also have \( t_u \Rightarrow_x t_k \). Now suppose \( v_{\text{id}}[t_i] < v_{\text{id}}[t_u] \). In this case, \( t_u \) writes \( x \) with version number \( <v_{\text{id}}[t_u], 0 > \). We now show that \( t_k \) writes \( x \) with version number \( <v_{\text{id}}, l > \), where \( v_{\text{id}} \geq v_{\text{id}}[t_u] \) and \( l \geq 1 \). Recall that there is a copy \( x_s \) that \( t_u \) accesses and later \( t_k \) writes. When \( t_u \) accesses \( x_s \), site \( s \) must already have, or it must immediately install, a view whose view-id is greater than or equal to \( v_{\text{id}}[t_u] \). Thus, when \( t_k \) later writes \( x_s \), site \( s \) must have a view \( v \) with view-id \( v_{\text{id}} \geq v_{\text{id}}[t_u] \). From our rule for write operations, \( t_k \) must have executed in view \( v \). Thus \( t_k \) writes \( x \) with version number \( <v_{\text{id}}, l > \) for some \( l \geq 1 \), which is greater than \( <v_{\text{id}}[t_u], 0 > \), the version number assigned by \( t_u \). Hence, \( t_u \Rightarrow_x t_k \).

(2) Suppose that \( t_u \) indirectly reads \( x \) from \( t_i \). There must be a sequence of update transactions \( t_{u_1}, t_{u_2}, \ldots, t_{u_{n-1}} \) such that \( t_u \) directly reads \( x \) from \( t_{u_{n-1}} \), \( \ldots, t_{u_2} \) directly reads \( x \) from \( t_{u_1} \), \( t_{u_1} \) directly reads \( x \) from \( t_i \). Since \( t_i \Rightarrow_x t_k \), by case (1), we have \( t_{u_1} \Rightarrow_x t_k \). Since \( t_{u_1} \Rightarrow_x t_{k_1} \), by case (1), we have \( t_{u_2} \Rightarrow_x t_k \). It is now clear that a simple induction shows that \( t_u \Rightarrow_x t_k \). □

We can now prove that \( SG[L] \) partially satisfies the third requirement of \( 1-SG[L] \).
Lemma 4.6: If \( t_j \text{ reads}_x \text{ from } t_i \), \( t_i \mathrel{\Rightarrow}_x t_k \), and \( t_j \) and \( t_k \) are user transactions executing in the same view, then \( SG[L] \) has an edge from \( t_j \) to \( t_k \).

Proof: \( t_j \) reads \( x \) and \( t_k \) writes \( x \) in the same view \( v \). By the quorum relation 
\[
q_r[x,v] + q_w[x,v] > n[x,v],
\]
there must be a copy \( x_s \) that \( t_j \) accesses and \( t_k \) writes. There are two cases to consider.

(1) Suppose \( t_j \) directly reads \( x \) from \( t_i \). Since \( t_i \mathrel{\Rightarrow}_x t_k \), then by the definition of \( \mathrel{\Rightarrow}_x t_k \) writes \( x_s \) with a greater version number than the one \( t_i \) assigns to the copies of \( x \). By Lemma 4.2, the version number of a copy never decreases. Since \( t_j \) directly reads the value of \( x \) written by \( t_i \) and not \( t_k \), then it must be that \( t_j \) accesses \( x_s \) before \( t_k \) writes it. Hence \( SG[L] \) has an edge from \( t_j \) to \( t_k \).

(2) Suppose \( t_j \) indirectly reads \( x \) from \( t_i \). Then there must be an update transaction \( t_u \) such that \( t_j \) directly reads \( x \) from \( t_u \) and \( t_u \) reads \( x \) from \( t_i \) (directly or indirectly). Since \( t_i \mathrel{\Rightarrow}_x t_k \), then, by Lemma 4.5, \( t_u \mathrel{\Rightarrow}_x t_k \). Since \( t_j \) directly reads \( x \) from \( t_u \), then, by case (1), \( SG[L] \) has an edge from \( t_j \) to \( t_k \).

\[\square\]

Lemmas 4.1, 4.3 and 4.6 show that we can extend \( SG[L] \) into \( 1 - SG[L] \) by adding only the following edges:

1. If \( t_i \mathrel{\Rightarrow}_x t_j \), and \( t_i \) and \( t_j \) are user transactions executing in different views, then add an edge from \( t_i \) to \( t_j \). These are write edges.

2. If \( t_j \text{ reads}_x \text{ from } t_i \), \( t_i \mathrel{\Rightarrow}_x t_k \), and \( t_j \) and \( t_k \) are user transactions executing in different views, then add an edge from \( t_j \) to \( t_k \). These are reads – before edges.

So \( SG[L] \) is extended into \( 1 - SG[L] \) by adding write edges and reads – before edges.
edges between user transactions executing in different views.

Informally, the edges of $SG[L]$ capture a serialization order between any two user transactions that issue physically conflicting operations. However, with our threshold and quorum assignments, a read $x$ and a write $x$ executing in different views do not necessarily physically conflict. The same holds for two write $x$ operations. Thus, we extended $SG[L]$ into $1-\text{SG}[L]$, a graph that also orders user transactions issuing logically conflicting operations, by adding write and reads-before edges between user transactions executing in different views.

In the next section we prove that transactions executing in different views are serialized according to the view-ids of those views, i.e., if $t$ executes in $v$ with view-id $v\_id$ and $t'$ executes in $v'$ with view-id $v'\_id$, where $v\_id < v'\_id$, then $t'$ is serialized after $t$ in the global serialization order.

4.3.2.2. Acyclicity of $1-\text{SG}[L]$

Since the log $L$ is CP-serializable, $SG[L]$ is acyclic. In this section we show that its extension $1-\text{SG}[L]$ is also acyclic. We first prove that if $SG[L]$ has an edge from transaction $t_i$ to transaction $t_j$ then $v\_id[t_i] \leq v\_id[t_j]$. We then extend this result to the edges of $1-\text{SG}[L]$.

Lemma 4.7: If $SG[L]$ has an edge from $t_i$ to $t_j$ then $v\_id[t_i] \leq v\_id[t_j]$.

Proof: The edges of $SG[L]$ are between transactions that execute physically conflicting operations. Let $op_i[x_s]$ and $op_j[x_s]$ be the two physically conflicting operations executed by $t_i$ and $t_j$. Since $SG[L]$ has an edge from $t_i$ to $t_j$, then $op_i[x_s] <_L op_j[x_s]$. Denote by $v_i$ the view of $s$ when $op_i[x_s]$ is executed, and by $v_{i-\text{id}}$
the view-id of $v_i$. There are two cases to consider depending on whether $t_i$ is a user or update transaction:

(1) $t_i$ is a user transaction. Therefore, when $op_i[x_s]$ is executed, $s$ must have a view $v_i$ with $v_i.id = v.id[t_i]$. When $op_j[x_s]$ is executed, $s$ must have a view $v_j$ with view-id $v_j.id$ such that $v_j.id \leq v.id[t_j]$. Since $op_i[x_s] <_L op_j[x_s]$, $v_i$ is installed at $s$ before $v_j$. But a site installs views with increasing view-ids, therefore $v_i.id \leq v_j.id$. Hence, $v.id[t_i] \leq v.id[t_j]$.

(2) $t_i$ is an update transaction. When $op_i[x_s]$ is executed, $s$ must have a view $v_i$ with $v_i.id \leq v.id[t_i]$. If $v_i.id = v.id[t_i]$, then the proof of case (1) can be applied to show that $v.id[t_i] \leq v.id[t_j]$. If $v_i.id < v.id[t_i]$, then, immediately after the execution of $op_i[x_s]$, $s$ installs a view $v'_i$ with view-id $v'_i.id \geq v.id[t_i]$. Since $op_i[x_s] <_L op_j[x_s]$, $t_j$ is either the update transaction triggered by the installation of $v'_i$ by $s$, or a user transaction or an update transaction that operates on $s$ after the installation of $v'_i$. In the first case, $v[t_j] = v'_i$ and $v.id[t_j] = v'_i.id \geq v.id[t_i]$. The second case, is similar to the proof of (1). When $op_j[x_s]$ is executed, $s$ must have a view $v_j$ with view-id $v_j.id \leq v.id[t_j]$ and $v_j.id \geq v'_i.id$. Combining inequalities results in $v.id[t_i] \leq v'_i.id \leq v_j.id \leq v.id[t_j]$.

We now extend the previous lemma to edges in $\bar{SG}[L]$.

Lemma 4.8: If $\bar{SG}[L]$ has an edge from $t_i$ to $t_j$ then $v.id[t_i] \leq v.id[t_j]$.

Proof: If the edge $<t_i, t_j>$ is in $SG[L]$, Lemma 4.7 implies $v.id[t_i] \leq v.id[t_j]$. If the edge $<t_i, t_j>$ is not in $SG[L]$ then it is either a write edge or a reads−before edge between user transactions that execute in different views. If $<t_i, t_j>$ is a write edge, then by definition $t_i \Rightarrow^x t_j$. Thus, the version
number assigned by $t_i$, $<v_{id}(t_i), k_i>$, is less than the one assigned by $t_j$, $<v_{id}(t_j), k_j>$, and therefore $v_{id}(t_i) \leq v_{id}(t_j)$.

Hence, we only have to consider the case where $<t_i, t_j>$ is a reads-before edge between two user transactions. In this case, there must be a transaction $t_h$ such that $t_i$ reads $x$ from $t_h$ and $t_h \Rightarrow x t_j$. There are two possible cases.

1. $t_i$ and $t_h$ execute in the same view, i.e., $v_{id}(t_i) = v_{id}(t_h)$. Since $t_h \Rightarrow x t_j$ then $v_{id}(t_h) \leq v_{id}(t_j)$. Therefore $v_{id}(t_i) \leq v_{id}(t_j)$.

2. $t_i$ and $t_h$ execute in different views. Thus, $t_i$ indirectly reads $x$ from $t_h$, i.e., there must be an update transaction $t_u$ such that $t_i$ directly reads $x$ from $t_u$ and $t_u$ reads $x$ from $t_h$. Since user transaction $t_i$ directly reads $x$ from $t_u$, we have $v_{id}(t_u) = v_{id}(t_i)$. Since update transaction $t_u$ reads $x$ from $t_h$ and $t_h \Rightarrow x t_j$, then by Lemma 4.5, $t_u \Rightarrow x t_j$. Thus, $v_{id}(t_u) \leq v_{id}(t_j)$ and $v_{id}(t_i) \leq v_{id}(t_j)$. □

We can now show that $1-SG[L]$ is acyclic, and therefore $L$ is one-copy serializable.

**Theorem 4.9**: $1-SG[L]$ is acyclic.

**Proof**: For contradiction, suppose $1-SG[L]$ has a cycle. From Lemma 4.7, it is clear that all transactions in this cycle must have the same view-id, and hence execute in the same view. Since $SG[L]$ is acyclic, the cycle in $1-SG[L]$ has at least one edge that is not in $SG[L]$. Thus it has at least one reads-before or write edge between two transactions executing in different views, a contradiction.
Hence, $1 - SG[L]$ is acyclic. □

4.4. Optimizations

The basic protocol, presented in Section 4.2, is sufficient to ensure one-copy serializability. However, its implementation can include several optimizations to increase the protocol's efficiency. In this section we describe two types of optimizations: those that increase data availability, and those that reduce the costs of update transactions.

4.4.1. Increasing Data Availability

In the basic protocol, the availability of an object $x$ is limited by the fact that a user transaction executing in a view $v$ is only allowed to touch (access or write) copies that reside on sites with the same view $v$. Such a transaction is aborted if it cannot touch a quorum of copies residing on sites with this view. In this section we propose two approaches to relax this restriction. These optimizations are easy to integrate with the basic protocol.

4.4.1.1. Relaxing the Read and Write Rules

With the basic protocol, a user transaction $t$ executing in some view can only access and write copies residing on sites with the same view. To increase data availability, we first relax this rule for write operations, while maintaining the restriction on read operations. We then discuss relaxing the restriction on read operations.
Relaxed Write Rule: A user transaction $t$ executing in view $v$ writes an object $x$ by writing $q_w[x,v]$ copies of $x$ residing on sites in $v$ and with a view whose view-id is less than or equal to $v\_id[t]$. Once a copy is written by $t$, it rejects all operations issued by user transactions executing in views with view-ids less than $v\_id[t]$. Furthermore, later views installed at site $s$ must have a view_id greater than or equal to $v\_id[t]$.

Note that a read operation of object $x$ executed by $t$ must still access $q_r[x,v]$ copies residing on sites in $v$, with view $v$. Our protocol, with the relaxed write rule still ensures one-copy serializability. The proof of correctness follows the one in Section 4.3, with some minor modifications. In particular, the proofs of Lemmas 4.1 to 4.4 do not change. The proof of Lemma 4.5 needs a slight modification that we leave to the reader. The proof of Lemma 4.6 does not change, and the proof of Lemma 4.7 is slightly modified as follows.

**Lemma 4.10:** If $SG[L]$ has an edge from $t_i$ to $t_j$ then $v\_id[t_i] \leq v\_id[t_j]$.

**Proof:** The edges of $SG[L]$ are between transactions that execute physically conflicting operations. Let $op_i[x_s]$ and $op_j[x_s]$ be the two physically conflicting operations executed by $t_i$ and $t_j$. Denote by $v_i$ the view of $s$ when $op_i[x_s]$ is executed, and by $v_i\_id$ the view-id of $v_i$ ($v_j$ and $v_j\_id$ are similarly defined). Since $SG[L]$ has an edge from $t_i$ to $t_j$, then $op_i[x_s] <_L op_j[x_s]$. But a site installs views with increasing view-ids, therefore $v_i\_id \leq v_j\_id$. Irrespective of whether $t_i$ is a user or update transaction, our new rule ensures that when $op_i[x_s]$ ($op_j[x_s]$) is executed, $v_i\_id \leq v\_id[t_i]$ ($v_j\_id \leq v\_id[t_j]$). If $op_i[x_s]$ is an access operation issued by a user transaction, then $v_i\_id = v\_id[t_i]$. Hence, $v\_id[t_i] \leq v\_id[t_j]$. 
Assume $v_{i}.id < v_{.id}[t_{i}]$, hence $op_{i}[x_{s}]$ is either a write operation or an access issued by an update transaction. In both cases, after the execution of $op_{i}[x_{s}]$, no operation is executed until $s$ installs a view $v'$ with view-id $v'.id$ such that $v_.id[t_{i}] \leq v'.id$. Since $t_{j}$ is either the update transaction triggered by the installation of $v'$ by $s$, or a user transaction or an update transaction that operates on $s$ after the installation of $v'$, then $v'.id \leq v_{j}.id$. Hence $v_.id[t_{i}] \leq v_{j}.id$. 

Finally the proofs of Lemma 4.8 and Theorem 4.9 do not change.

Relaxing the read rule introduces a tradeoff between availability and cost. To increase data availability, we could allow a user transaction executing in view $v$ to read an object $x$ by accessing copies that reside on sites with views other than $v$, even if this object is not read accessible in $v$. However, each read $x$ must now access $A_{r}[x]$ copies of $x$ (instead of $q_{r}[x,v]$ copies that reside on sites in $v$ with view $v$).

Alternative Read Rule: A user transaction $t$ reads an object $x$ by accessing $A_{r}[x]$ copies of $x$ residing on sites with a view whose view-id is less than or equal to $v_.id[t]$. Once a copy is accessed by $t$, it rejects all write operations issued by user transactions executing in views with view-ids less than $v_.id[t]$. Furthermore, later views installed at site $s$ must have a view-id greater than or equal to $v_.id[t]$.

With the alternative read rule, the proofs of Lemmas 4.1 to 4.6 essentially remain the same. A minor modification is needed to the last paragraph of the proof of Lemma 4.10 above. If $op_{i}[x_{s}]$ is an access operation, then $op_{j}[x_{s}]$ must be a write operation (since $op_{i}[x_{s}]$ and $op_{j}[x_{s}]$ are conflicting operations). By our rules,
\(\text{op}_j[x_s]\) cannot be executed until \(s\) installs view \(v'\) where \(v\_id[t_i] \leq v'\_id\), hence \(v\_id[t_i] \leq v\_id[t_j]\).

The proof of Lemma 4.8 has to be modified as follows:

**Lemma 4.11:** If \(1 - \text{SG}[L]\) has an edge from \(t_i\) to \(t_j\) then \(v\_id[t_i] \leq v\_id[t_j]\).

**Proof:** The original proof holds except in the case where \(<t_i, t_j>\) is a \textit{reads - before} edge and \(t_i\) uses the alternative read rule to read a value written by some \(t_h\), where \(t_h \Rightarrow_x t_j\). There are two cases to consider:

1. Suppose that \(t_i\) \textit{directly reads} \(x\) \textit{from} \(t_h\). User transaction \(t_i\) uses the alternative read rule and accesses \(A_r[x]\) copies of \(x\), and user transaction \(t_j\) writes \(q_w[x,v] \geq A_w[x]\) copies of \(x\) (for some \(v\)). Since \(A_r[x] + A_w[x] > n[x]\) (the total number of copies of \(x\)), there must be at least one common copy \(x_s\) that \(t_i\) accesses and \(t_j\) writes. Since \(t_h \Rightarrow_x t_j\) then, by the definition of \(\Rightarrow_x\), the copies of \(x\) that \(t_h\) writes have a smaller version number than the copies written by \(t_j\). By Lemma 4.2, the version number of a copy never decreases. Since \(t_i\) \textit{directly reads} \(x\) \textit{from} \(t_h\) and not from \(t_j\), it must be that \(t_i\) accesses \(x_s\) before \(t_j\) writes it. But once \(t_i\) accesses \(x_s\) using the alternative read rule, site \(s\) rejects all write operations issued by user transactions executing in views with view-ids less than \(v\_id[t_i]\). Hence, \(v\_id[t_i] \leq v\_id[t_j]\).

2. Suppose that \(t_i\) \textit{indirectly reads} \(x\) \textit{from} \(t_h\). Then there must be an update transaction \(t_u\) such that \(t_i\) \textit{directly reads} \(x\) \textit{from} \(t_u\) and \(t_u\) \textit{reads} \(x\) \textit{from} \(t_h\) (directly or indirectly). Since \(t_h \Rightarrow_x t_j\), then, by Lemma 4.5, \(t_u \Rightarrow_x t_j\). Since \(t_i\) \textit{directly reads} \(x\) \textit{from} \(t_u\) using the alternative read rule, and \(t_u \Rightarrow_x t_j\), then, by case (1), \(v\_id[t_i] \leq v\_id[t_j]\). \(\square\)
4.4.1.2. Using Multi-Versions

Multi-version databases are widely known to increase data availability [Reed83, Herl87]. We can easily integrate multi-versions to our protocol by simply associating with each copy a sequence of versions, each corresponding to a different view (and view-id). The versions of a copy are ordered by their associated view-ids. Furthermore, each site maintains a sequence of views installed at that site, with their view-ids and corresponding quorums. For a user transaction \( t \) to execute, it chooses one of these views \( v \), and we say \( t \) executes in \( v \) (note that \( v \) does not have to be the most recent view installed at the initiating site). The latest view installed at site \( s \) is considered to be the view of \( s \).

With multi-version databases, an update transaction \( t_u \) installing a new view \( v[t_u] \), with view-id \( v_id[t_u] \) is executed as follows:

**Update Transaction Rule:** For each object \( x \) that is read accessible in \( v[t_u] \):

1. \( t_u \) accesses \( A_r[x] \) copies of \( x \),
2. each accessed site rejects any write \( x \) by a user transaction executing in a view with a view-id less than \( v_id[t_u] \).
3. for each accessed copy, \( t_u \) accesses the version with the highest view-id less than or equal to \( v_id[t_u] \),
4. \( t_u \) then determines \( vnmax \), the maximum version number of the accessed versions, and reads the accessed version associated with \( vnmax \), and
5. \( t_u \) writes the value read into the \( v_id[t_u]^{th} \) version of the local copy of \( x \). The associated version number is \( <v_id[t_u], l> \), where \( l \geq 0 \) is the smallest integer that is greater than or equal to \( vnmax \).
Note that, unlike the single version case, some sites accessed by $t_u$ may have a view whose view-id is greater than $v_{id}(t_u)$.

With multi-version databases, a user transaction $t$ executing in view $v$ must observe the following write rule.

**Multi-Version Write Rule:** If object $x$ is write accessible in view $v$:

1. $t$ accesses $q_w[x,v]$ copies of $x$ residing on sites in $v$,
2. for each such copy, $t$ accesses the version with the highest view-id less than or equal to $v_{id}(t)$,
3. $t$ determines $vn_{max}$, the maximum version number of the accessed versions, and
4. $t$ writes the $v_{id}(t)^{th}$ version of all the selected copies and updates their version numbers to $<v_{id}(t), l>$, where $l \geq 1$ is the smallest integer such that $<v_{id}(t), l>$ is greater than $vn_{max}$.

A user transaction $t$ executing in a view $v$ must observe one of the following read rules, which represent another tradeoff between cost and availability:

**Multi-Version Read Rule 1:** If object $x$ is read accessible in view $v$:

1. $t$ accesses $q_r[x,v]$ copies of $x$ residing on sites in $v$ and with a view whose view-id is greater than or equal to $v_{id}(t)$,
2. for each such copy, $t$ accesses the $v_{id}(t)^{th}$ version,
3. $t$ determines $vn_{max}$, the maximum version number of the accessed versions, and reads the accessed version associated with $vn_{max}$.

With the second read rule, a user transaction $t$ executing in view $v$ can read $x$ even
if $x$ is not read accessible in $v$.

**Multi-Version Read Rule 2:** $t$ reads $x$ by following the first four steps of an update transaction.

The multi-version read rules represent a tradeoff between availability and cost. Using read rule 1, a transaction $t$ executing in $v$ can read $x$ by accessing $q_r[x,v]$ copies of $x$ that reside on sites that have installed view $v$ (or a view with a higher view-id). In contrast, if $t$ reads $x$ using read rule 2, then it can access copies of $x$ residing on any sites regardless of their views; however, $t$ must access $A_r[x]$ copies (note that $A_r[x] \geq q_r[x,v]$).

Multi-versions increase data availability. To illustrate this increase in availability (with respect to single version databases), we reconsider the example discussed in Section 3. Recall that object $x$ has copies residing on sites $s_1$, $s_2$ and $s_3$, and object $y$ has copies residing on sites $s_2$, $s_3$ and $s_4$. Furthermore, $A_r[x] = A_w[x] = A_r[y] = A_w[y] = 2$, and initially in view $v_0 = \{s_1,s_2,s_3,s_4\}$, $q_r[x,v_0] = q_r[y,v_0] = 1$ and $q_w[x,v_0] = q_w[y,v_0] = 3$.

Recall that in the single version case, after the network partitions into $P_1 = \{s_1,s_2\}$, and $P_2 = \{s_3,s_4\}$, the sites in $P_1$ have two options: (1) they either keep the old view $v_0 = \{s_1,s_2,s_3,s_4\}$, thus retaining the ability to read $x$ and $y$, but loosing the ability to write $x$, or (2) they install a new view $v_1 = \{s_1,s_2\}$ (with read quorum $q_r[x,v_1] = 1$, and $q_w[x,v_1] = 2$), thus retaining the ability to read and write $x$, but loosing the ability to read $y$. In contrast, with multi-versions, the sites in $P_1$ can retain both the ability to execute transactions that read and write $x$, as well as transactions that read $y$. They install the new view $v_1 = \{s_1,s_2\}$, and
execute transactions as follows. A read $x$ or write $x$ transaction issued in partition $P_1$ can be executed in the new view $v_1$, (e.g., write $x$ writes the $v_{1}^{th}$ version of \( q_{a}[x,v_1] \) copies of $x$ in $v_1$). A read $y$ transaction issued in partition $P_1$ can be executed in the old view $v_0$ (by reading the $v_{0}^{th}$ version of \( q_{r}[y,v_0] \) copies of $y$ in $v_0$).

Note that with multi-versions, in $P_1$, a transaction can read and write $x$, by executing in new view $v_1$, while another transaction can read $x$ and $y$, by executing in view $v_0$. At the same time, symmetrically, $P_2$ can execute transactions that read and write $y$, and others that read $x$ and $y$. In a single version database, no accessibility thresholds and quorum assignments allow both $P_1$ and $P_2$ to execute these transactions.

Multi-version databases increase data availability, but storing all the versions of each copy can be expensive. To reduce costs, the database may store only the most recent versions of each copy (more than one version but not necessarily all previous versions). In this case, some user transactions may need to access versions that were not saved, and thus be aborted. This intermediate scheme achieves lower availability than multi-versions, at a lower cost. Wright studied the degree of availability provided by the class conflict protocol [Wrig84, Skee84] when two or more versions of a copy are stored, and showed an increase in data availability over the single version case, without incurring the expensive overhead of storing all versions of a multi-version database. In ISIS [Jose86], each copy keeps the two most recent versions for recovery purposes; using our protocol, an increase in data availability can be achieved using these two versions.
4.4.2. Reducing the Cost of Update Transactions

Various modifications can be made to optimize update transactions depending on the specific application. In this section, we present four such optimizations. We first reduce the redundancy of read operations executed by the different update transactions installing the same view at different sites. Second, we reduce the granularity of update transactions by associating views with copies instead of with sites. We also show that this optimization results in an increase in data availability. Third, we briefly discuss methods for reducing the communication costs of access operations by storing copies as logs of operations. Finally, we consider an optimization that reduces the cost of update transactions for databases that suffer mainly from site failures, and not partitioning failures.

4.4.2.1. Reducing the Number of Update Transactions

Our first optimization reduces the redundancy of read operations executed by update transactions installing the same view at different sites. Consider a set of sites installing the same new view. One site initiates the new view, and several others inherit it. For each read accessible object $x$, we previously required that all sites installing this new view independently access $A_r[x]$ copies of $x$ to update their local copy of $x$. This is clearly redundant and expensive: a single site could do the $A_r[x]$ accesses to update its local copy of $x$, and then propagate this copy to all other sites installing the same view (we used a similar approach in [Abba85]). If the new view to be installed contains $l$ copies of $x$, this optimization reduces by a
factor of \( l \) the "read cost" of updating \( x \) during installation of this view.

4.4.2.2. Reducing the Granularity of Update Transactions

The second optimization reduces the granularity of update transactions. So far, we associated a view with each site. Thus to change the view of a site \( s \), the update transaction initiated at \( s \) must update all the local copies of read accessible objects in the new view. As Herlihy [Herl87] pointed out, reducing the granularity of update transactions may have several advantages, and the quorum consensus protocol performs updates on a per object basis, rather than per site. We can do the same simply by associating a view with each copy of an object instead of with each site. (This is equivalent to considering each copy as residing on a virtual site of its own.) Now each copy \( x_s \) has its own view consisting of a set of copies rather than a set of sites. To change the view of a copy \( x_s \), the update transaction initiated at \( s \) must only update the value of \( x_s \).

In addition to reducing the granularity of update transactions, assigning views to copies instead of sites also increases the availability of the database. This is illustrated by the example we considered in Section 3. Recall that \( x \) has three copies residing on sites \( s_1, s_2 \) and \( s_3 \), and \( y \) has three copies residing on \( s_2, s_3 \) and \( s_4 \). Furthermore, \( A_r[x] = A_w[x] = A_r[y] = A_w[y] = 2 \). Initially, all copies of \( x \) and \( y \) have the same initial view \( v_0 = \{x_{s_1}, x_{s_2}, x_{s_3}, y_{s_2}, y_{s_3}, y_{s_4}\} \). After the network partitions into \( P_1 = \{s_1, s_2\} \), and \( P_2 = \{s_3, s_4\} \), instead of requiring all copies in \( P_1 \) to either keep their old view \( v_0 \) (thus retaining the ability to read \( x \) and \( y \), but loosing the ability to write \( x \)), or to form a new view \( v_1 = \{x_{s_1}, x_{s_2}, y_{s_2}\} \) containing all copies residing on sites in \( P_1 \) (thus retaining the ability to read and write \( x \),
but loosing the ability to read y), reducing the granularity gives us a third alternative. Copy \( y_{s_2} \) may keep its old view \( v_0 \), while copies \( x_{s_1} \) and \( x_{s_2} \) install the new view \( v_1 \). Thus a read \( x \) or write \( x \) transaction issued in partition \( P_1 \) can be executed in the new view \( v_1 \) (e.g., write \( x \) writes \( q_{v_1}[x,v_1] \) copies of \( x \)). A read \( y \) transaction issued in partition \( P_1 \) can be executed in the old view \( v_0 \), by reading \( q_r[y,v_0] \) copies of \( y \). Hence, if we associate views with copies, instead of with sites, we allow read \( x \), write \( x \), and read \( y \) transactions to execute in \( P_1 \). However, we note that a transaction that reads both \( x \) and \( y \) would not be able to execute in \( P_1 \), since this transaction must choose to execute in \( v_0 \) or in \( v_1 \) (but cannot execute in both).

4.4.2.3. Reducing Communication Costs of Update Transactions

The third optimization reduces the communication costs of the access operations executed by update transactions. A copy can either be stored explicitly by writing out its complete state, or by maintaining a log of operations executed on that copy. If objects are "large", i.e., the complete description of their state is large, the communication costs incurred in transferring the whole state of a copy can be expensive. To reduce this communication cost, a sending site can send a log of all the operations on that object that the receiving site has missed. Suppose a site \( s \) installs a view with view_id \( v_{-id} \), and must update copy \( x_s \), last written by transaction \( t \). Then site \( s \) needs to receive only the write operations executed on \( x \) in views with view-ids greater than \( v_{-id}[t] \), but less than \( v_{-id} \).
4.4.2.4. Optimizing Update Transactions after Site Failures

Finally, we introduce an optimization for reducing the cost of update transactions following site failures. A view $v'$ installed after some site failures is usually just a subset of a previous view $v$. This new view may retain the read quorums of the previous view for most objects $x$. If $A_r[x]$ copies of $x$ still reside on sites with the previous view $v$, the cost of updating an object $x$ (when installing $v'$) can be reduced as follows.

Let $s$ be a site that executes an update transaction to change its view from $v$ to $v'$. For each object $x$ that does not change its read quorum, site $s$ sends messages to $A_r[x]$ copies of $x$ that reside on sites in $v$. Each site $p$ that receives such a message, and whose current view is still $v$, or whose view was $v$ and was changed to $v'$ (with no other views in between), sends an acknowledgement to $s$, and immediately tries to inherit $v'$ (if its view is not already $v'$). If $p$'s current view does not satisfy either of these two cases, $p$ ignores that message. If $s$ receives $A_r[x]$ acknowledgements, then it does not have to change its value of $x_S$. By not requiring the actual accessing and transmission of $A_r[x]$ copies, the communication cost of the update transaction is significantly reduced. Informally, we note that since the new view $v'$ is a subset of the previous view $v$, for any read accessible object $x$ that retains the read quorums of $v$ in $v'$, all read operations of $x$ executed in $v'$ are serialized after all writes of $x$ in $v$, and hence this optimization ensures one-copy serializability.
4.5. Comparison with Other Work

In the virtual partitions protocol protocol, whenever the communication topology changes, a two-phase protocol is first executed to ensure consistency of views. Then, an update operation is initiated that updates the different copies to ensure the consistency of data. The cost of a separate view management protocol is a disadvantage of this method. We observe that while the update protocol is necessary to ensure consistency of data, the two-phase view management protocol is redundant.

With the virtual partitions protocol, the write accessibility thresholds must satisfy the additional requirement that $2A_r[x] > n[x]$ (in addition to threshold requirement (1)). With this extra requirement only one partition could write an object $x$ in any network configuration (the partition with the majority of copies of $x$). Furthermore, the read and write quorums are fixed to $q_r[x,v] = 1$ and $q_w[x,v] = n[x,v]$ for all views $v$. The accessibility thresholds protocol allows more flexibility in the choice of quorums: each view may have different quorum assignments for the same object $x$, as long as they satisfy quorum relations (2), (3), (4) and (5). This, in addition to the flexibility in choosing the accessibility thresholds $A_r[x]$ and $A_w[x]$ and the optimizations outlined in Section 4.4, provides the database designer with a larger degree of freedom in deciding the cost of operations, and the degree of data availability.

The accessibility thresholds protocol also provides a higher degree of availability and flexibility than the protocols proposed in [Giff79, Eage83]. Gifford's protocol associates with each object $x$ fixed read and write quorums $Q_r[x]$ and $Q_w[x]$,
where $Q_r[x] + Q_w[x] > n[x]$ and $2Q_w[x] > n[x]$. This is a special case of our protocol where all the sites have the same view $v$ that includes all the sites of the database, and where $q_r[x,v] = A_r[x] = Q_r[x]$ and $q_w[x,v] = A_w[x] = Q_w[x]$. In contrast to our protocol, if the network partitions, at most one partition can write $x$ (since $2Q_w[x] > n[x]$).

Eager and Sevick [Eage83] introduced the missing write protocol, which is an extension of the Gifford protocol. Initially, transactions (called normal mode transactions) use a read-one, write-all protocol. When a transaction cannot execute due to a failure, it switches to failure mode, and for each object it uses read and write quorums $Q_r[x]$ and $Q_w[x]$, as in Gifford's protocol. To ensure serializability, all failure mode transactions leave a missing write token at all copies they access or write. When a normal mode transaction encounters a missing write token, it switches to failure mode. This protocol is also a special case of our protocol, where all views include all the sites of the database. Initially, all transactions execute in view $v_n$ with quorums $q_r[x,v_n] = 1$ and $q_w[x,v_n] = n[x]$. When a site detects a failure, it installs view $v_f$ with quorums $q_r[x,v_f] = A_r[x] = Q_r[x]$ and $q_w[x,v_f] = A_w[x] = Q_w[x]$.

Our protocol reduces the number of physical operations executed, compared to the protocols in [Giff79, Eage83]. For example, consider an object $x$ with $n[x]$ copies. Let $Q_r[x]$ and $Q_w[x]$ be the quorums associated with $x$ in any of the protocols in [Giff79, Eage83]. To achieve the same level of availability as these protocols do, we set the thresholds and quorums of our protocol to $A_r[x] = q_r[x,v] = Q_r[x]$ for all $v$, and $A_w[x] = Q_w[x]$. With these threshold and quorum assignments, any
read or write operation on $x$ that is executable using quorum protocols is also executable with our protocol. Let $P$ be a partition where all sites have view $v = P$. To write an object $x$ in $P$, the protocols in [Giff79, Eage83] require writing $Q_w[x] > n[x] - Q_r[x]$ copies, compared to writing $q_w[x,v] > n[x,v] - q_r[x,v]$ copies with our protocol. Note that $n[x,v] \leq n[x]$ and $q_r[x,v] = Q_r[x]$.

With our protocol, as with the virtual partitions protocol, it is never necessary for a read operation to physically access more than one copy, not even if the database partitions. This is in contrast to the protocols in [Giff79, Eage83], where each read is required to access more than one copy, in order to execute write operations in the presence of failures. A write operation on an object $x$ cannot require the physical writing of more than $n[x] - f$ copies, if it is supposed to execute despite the inaccessibility of $f$ copies. For this case, the protocol in [Giff79] requires that read operations always physically access at least $f + 1$ copies. This is improved upon in [Eage83] where these expensive read operations are necessary only when failures occur. With our protocol, it is never necessary to incur this high cost, even if partitioning occurs.

To illustrate the possible choices of thresholds and quorums with our protocol, consider an object $x$ with $n[x] = 8$. Let $P$ be a partition where all sites have view $v = P$, and $n[x,v] = 6$. By setting the accessibility thresholds to $A_r[x] = 5$ and $A_w[x] = 4$ (thus satisfying threshold relation (1)), $x$ is read and write accessible in $P$. The database designer has the following three possible choices for the read and
write quorums of $x$ in partition $P^\dagger$:

<table>
<thead>
<tr>
<th>Quorums</th>
<th>Choice I</th>
<th>Choice II</th>
<th>Choice III</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_{r[x,v]}$</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$q_{w[x,v]}$</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

All three choices satisfy the quorum relations (2), (3), (4) and (5). With choices I, II, and III a read operation on $x$ physically accesses 1, 2, or 3 copies, respectively.

With the protocols [Giff79, Eage83], to allow the writing of $x$ in $P$ one can set either $Q_w^r[x] = 6$, in which case a read $x$ has to access 3 copies (and a write $x$ writes 6 copies), or $Q_w^r[x] = 5$, in which case a read $x$ has to access 4 copies (and a write $x$ writes 5 copies). Note that with these protocols writing only four copies of $x$ (as in our choice III) is not allowed since four copies are not a majority of copies of $x$.

Note that whenever views change, our approach requires the execution of update transactions. With Gifford’s protocol there are no update transactions, but, as illustrated above, user transactions can be more expensive than with our method. We assume that the network topology does not change often, and hence update transactions are rare with respect to user transactions. This justifies a higher cost for update transactions, and a lower cost for user transactions.

For multi-version databases, Herlihy [Herl87] presents a generalization of Gifford’s quorum protocol called the quorum consensus protocol. Each copy of an object has several versions ordered by levels. Each object has a quorum assignment table with a sequence of quorum assignments. Each quorum assignment

$\dagger$ This choice can be made by the initiator of the view $v$, and then imposed on the other sites that inherit this view (by passing the chosen quorums via the update transaction the initiator executes).
corresponds to a level, and for a transaction to execute, it chooses a level, and uses the quorum assignments associated with that level to execute operations. The quorum assignments are restricted to satisfy the quorum intersection invariant: "Each write quorum associated with level l must intersect with each read quorum associated with a level greater than or equal to l".

One of the main advantages of the quorum consensus protocol is that it adjusts "lazily" to changes in the network on a per object basis. As we saw in the Section 4.2.3 and Section 4.4.2, our protocol can retain both advantages by using "on-demand" view tracking, i.e., views are changed only when some "high priority" transactions can no longer execute in the previous view, and, views can be associated with copies, thus allowing update transactions to be executed on a per object basis, only when necessary.

There are tradeoffs between our protocol and the quorum consensus protocol in terms of costs. The quorum consensus protocol is designed for multi-version databases. It must maintain a quorum assignment table and ensure that this table always satisfies the quorum intersection invariant. This overhead allows transactions to run at increasingly higher levels (by a process called inflation) without incurring update costs. However, to satisfy the quorum intersection invariant, read quorums must monotonically increase with respect to level number, thus making read operations more expensive at higher levels (thus, if failures are frequent, this protocol adapts more readily that our protocol; however, read operations become more expensive). Hence, objects eventually need to reduce the read quorums assigned to their higher levels. For this purpose, objects execute a pro-
cess called deflation, which is similar to our update transaction. One advantage of our protocol is that there is no need to maintain a quorum assignment table. Furthermore, when a site decides to install a new view (to execute transactions at a "higher level" in [Herl87] terminology), it can freely choose the read and write quorums associated with the new view, without being restricted by an assignment that must satisfy the quorum intersection invariant for a given quorum assignment table.

4.6. Commercial Distributed Databases

As of the mid 1980's, very few commercial databases supported data replication [Moh84, Car85]; however, several research projects and prototypes have been developed. One of the first distributed databases was $R^*$ [Lin84], which did not support replication and used a blocking commit protocol (operator intervention is required to resolve blocking) [Moh]. Distributed INGRES [Sto86] is a distributed database system that supports replication and provides two types of protocols for the management of replicated objects [Sto79]. The first protocol is a primary site protocol where a transaction updates the primary copy of an object, and then the updated information is propagated to other copies after the transaction commits. This approach may result in several types of inconsistencies and non-serializable executions [Sto79]. The second protocol is more complex and requires the writing of all copies residing on sites in a partition. Furthermore, information about the system configuration has to be propagated during transaction execution. In the DDM system [Good83], as well as in the SDD-1 system [Hame80], which were developed at CCA, protocols were implemented for managing replicated data
in the presence of site failures. These protocols use an approach very similar to the available copies protocol, which does not tolerate partitioning failures. Finally, at Tandem [Borr84, Tand87] full duplication is used to ensure fault-tolerance to any single failure. In this system every process has a backup process, and checkpointing is used to provide the backup process with the information needed in the event of a process failure.

Thus, current commercial databases are severely restricted in their fault-tolerance and the degree of data availability they provide, in particular when partitioning failures occur. However, some experimental systems like ISIS [Jose85, Jose86] have developed highly sophisticated and efficient mechanisms to reduce the communication costs of managing replicated data, e.g., the piggybacking update method for deferring physical write operations until their results are actually needed. Our protocol can easily use any of these methods to reduce communication costs.

4.7. Conclusion

In this chapter, we presented a new concurrency control protocol for reading and writing replicated data in spite of site and communication failures. The protocol uses views to ensure one-copy serializability. First the protocol ensures that in each view, all transactions are one-copy serializable (this is achieved by using intersecting read and write quorums). Then it ensures that all transactions executing in one view are serialized after all transactions executing in views with lower view-ids, i.e., transactions executing in different views are serialized according to view-ids (this is achieved by using the proper accessibility thresholds, and
the update transaction).

The choice of accessibility thresholds, quorums, and views gives a large degree of flexibility in determining the availability of objects and the costs of executing read and write operations. First, one can choose the read and write accessibility thresholds for each object. These thresholds determine the read and write availability of each object in all views (depending on the number of copies in those views). To increase the read availability of an object, the read accessibility threshold is decreased, and vice-versa. Second, for each new view installed during execution, one can choose a read and a write quorum for each object (that is read or write accessible in the new view). These quorums determine the costs of executing read and write operations for each object in each new view. Finally, the database designer may choose among several policies for deciding when to change views following changes in the network topology, and which sites to include in the new views. These view changing policies also determine the availability of different objects, and the cost of executing operations.
CHAPTER 5

The Group Paradigm

5.1. Introduction

In this chapter we present a paradigm for developing, describing and proving the correctness of concurrency control protocols for replicated databases in the presence site and communication failures, even when these failures lead to partitioning. Our approach is to hierarchically divide the problem of achieving one-copy serializability by introducing the notion of a "group" that is a higher level of abstraction than a transaction. A group is a set of transactions. Instead of dealing with the overall problem of serializing all transactions, our paradigm divides the problem into two simpler ones. Informally, one must first derive a local policy for each group that ensures a total order of all transactions in that group. This guarantees that each group, as a whole, transforms the database from one consistent state to another, and hence, each group can be viewed as a high level transaction. Then, one should develop a global policy that ensures a correct serialization of all groups. The serialization order of all groups, combined with the serialization of all transactions in a group, ensures one-copy serializability. A protocol is an instance of the paradigm if it partitions transactions into groups, where each group uses a local policy to serialize in that group, and a global policy that serializes groups.
To prove that any instance of the paradigm ensures one-copy serializability, we introduce a new model, the \textit{one-copy serialization group graph} that is particularly suited to our concept of a group as a high level transaction. The nodes of this graph are groups (i.e., sets of transactions), whereas in previous methods like the serialization graphs [Bern87], nodes are transactions. The use of groups, instead of transactions, as the basic unit simplifies the proofs, and highlights the properties required to ensure one-copy serializability. We prove that if transactions can be partitioned into groups such that all transactions in each group are one-copy serializable, and there is a corresponding acyclic one-copy serialization group graph, then all transaction executions are one-copy serializable.

We also describe and prove the correctness of several concurrency control protocols [Eage83, Herl87, Skee84], as well as the virtual partitions and the accessibility thresholds protocols, in terms of one model. We show that even though these protocols may appear to be unrelated, they are all instances of our group paradigm. This emphasizes the similarities between them, and hence simplifies their understanding.

In the next section, we present a simple example that motivates our paradigm. In Section 5.3 we present the group paradigm and in Section 5.4 we develop a correctness model that is particularly suitable for the group paradigm. In Section 5.5 an analysis of several concurrency control protocols is made that emphasizes the fact that they are instances of the group paradigm.
5.2. A Simple Motivating Example

Consider a database system consisting of a single object that supports a single update operation. The update operation reads the value of the object, and then writes it. In a replicated database, the object is implemented by a set of copies, and an update operation is implemented by a set of operations executed on these copies. For this replicated implementation to be correct, it must ensure one-copy equivalence, i.e. the object behaves as if it has only one copy in so far as the user can tell. For a non-replicated object, i.e., that is implemented by only one copy, if each update is atomically executed, then all updates are totally ordered and each update reads the value written by the preceding update. Since our correctness criteria is one-copy equivalence, any replicated implementation must ensure that there is a total order on all updates such that if some update \( u_j \) reads the value written by update \( u_i \), then no update \( u_k \) is ordered after \( u_i \) and before \( u_j \).

Consider the following approach for ensuring one-copy equivalence. We assume that an initial update, which is assigned an initial identifier (e.g. 0), is executed to initialize the object. Subsequently, every update \( u_i \) accesses a set of copies called its readset. Update \( u_i \) reads the value of the copy written by the update with the highest identifier. Update \( u_i \)'s identifier is chosen to be greater than this highest identifier. Finally, \( u_i \) writes a set of copies called its writeset. Each update is executed atomically by the use of a commit protocol [Ullm82, Skee82]. It can be shown that to ensure one-copy equivalence, the readset of every update must have at least one copy in common with the writeset of every other update [Giff79, Herl85]. One simple way to ensure this is to require all readsets and writesets to
contain at least a majority of copies.

However, requiring each update to be executed on a majority of copies may not be necessary, or even desirable. Consider, for example, an object that is implemented by nine copies. If four copies fail, then any update will access all five available copies (five copies are a majority of nine copies). This is only necessary for the first update operation executed after the failure, so as to ensure that it accesses at least one copy written by each update executed before the failure. However, for all subsequent updates, executed after the failure, accessing and writing five copies is not necessary since every update writes the same five copies. Accessing and writing three copies would be enough to ensure one-copy equivalence.

It may therefore be advantageous to use a more flexible approach to ensure a total order on all updates. First updates are divides into "groups", where each group may use a local policy to ensure a total order on all updates that are members of that group, e.g., a policy that requires a writeset to intersect with all readsets and writesets. Second, a total order is imposed on all groups. The total order of updates within each group, together with the total order of all groups, defines a total order on all updates.

In the above example, all updates executed before the failure of the four sites form a group $g_0$, where updates execute on a majority of copies (five copies). Subsequently when the four sites fail, a new group $g_1$ is formed of all updates executing on the five operational copies. Updates are executed on three of these copies only (three is a majority of five). All updates in $g_0$ form a total order, and all
updates in $g_1$ form a total order. Hence a total order is defined on all updates. However, a total order on all updates is not enough to ensure one-copy equivalence as we now show.

Let us consider a different case where the four failed sites do not fail but form a partition disconnected from the other five sites. All updates executing in that partition form a new group $g_2$. Assume that the first updates $u_1$ and $u_2$ executed in $g_1$ and $g_2$ read the same value written by $u_0$, the last update executed in $g_0$ (according to some total order on all updates in $g_0$). To preserve the one-copy equivalence of all updates, both $u_1$ and $u_2$ must immediately follow $u_0$. Hence, a total order on all groups must be constructed so that both $g_1$ and $g_2$ immediately follow $g_0$, which is impossible. It is thus clear that one cannot always find a total order of groups that is consistent with a total order on all updates executed in the different groups.

In this chapter we generalize the ideas discussed in this simple example. We present a paradigm for concurrency control protocols that is based on the concept of groups. Each group has a local policy to ensure a total order on all updates, or more generally the one-copy serializability of all transactions in a group. Furthermore, a global policy ensures a total order on all groups, such that all transaction executions are one-copy serializable.

5.3. The Paradigm

In this section, we first present the notion of a "group" as a high level transaction, and develop the necessary formal framework for the presentation of the group paradigm. The group paradigm for concurrency control protocols is then presented
in two components: the local policies, which ensure the serialization of all transactions in each group; and the global policy, which ensures the serialization of all groups in such a way that all transaction executions are one-copy serializable. In the next section we prove that protocols that adhere to this paradigm guarantee one-copy serializability.

5.3.1. Groups

Recall that a transaction is composed of a set of operations, which if executed alone, takes the database from one consistent state to another. We now propose a higher level of abstraction than the transaction: a group, which is a set of transactions. Formally, given a set of transactions $T$, a set of groups $\{g_1, g_2, \ldots, g_k\}$ partitions $T$ into disjoint sets, such that $\forall i, j \ g_i \cap g_j = \emptyset$ and $\bigcup_i g_i = T$. The execution of (all transactions in) a group should transform, as a whole, the database from one consistent state to another. It will be useful for the reader to think of groups as higher level transactions.

We now extend the definition of one-copy serialization to the set of transactions in a group (as opposed to the set of all transactions). In the standard model, it is assumed that a single transaction $t_{init}$ initializes the database. Note that transactions in a group may read values written by transactions belonging to several other groups. For a log $L$ ($L = (S_L, <_L)$) and group $g$, we introduce transaction $t_{init}^g$ that initializes the database from $g$'s point of view. Transaction $t_{init}^g$ contains all write operations executed by transactions not in $g$, but that write values read by transactions in $g$, i.e., $t_{init}^g = (S_{init}^g, <_{init}^g)$, where $S_{init}^g = \{ w[x] |$
where \( w_i(x) \) is issued by \( t_i \notin g \), and \( \exists t_j \in g \) such that \( t_j \text{reads}_x \text{from } t_i \), and \( ^g_{\text{init}} \leq \{ <op_i, op_j> | op_i, op_j \in S^g_{\text{init}} \text{ and } op_i <_L op_j \} \).

The sublog \( L_g \) is defined as a restriction of \( L \) to group \( g \). Sublog \( L_g \) is a partial order \( (S_g, <_L^g) \), where \( S_g \) is the set of all operations executed by transactions in \( g \), as well as the operations of \( t^g_{\text{init}} \), and \( <_L^g \) is the subset of \( <_L \) corresponding to operations in \( S_g \). Formally, \( S_g = \{ op_i | \text{where } op_i \text{ is issued by } t_i \in g \text{ or } op_i \in S^g_{\text{init}} \} \) and \( <_L^g = \{ <op_i, op_j> | op_i, op_j \in S_g \text{ and } op_i <_L op_j \} \).

A group \( g \) is **one-copy serializable** if \( L_g \) is equivalent to a serial one-copy log with \( t^g_{\text{init}} \) preceding all other transactions. A serial one-copy log defines a total order \( <_g \) over all the transactions in \( g \) as well as transaction \( t^g_{\text{init}} \). The **final value** of an object \( x \) with respect to a group \( g \) and total order \( <_g \) is defined as the value of \( x \) written by the last transaction (with respect to \( <_g \) that writes \( x \) in \( g \).

A group \( g \) is said to **write (read)** an object \( x \), if there is a transaction in \( g \) that writes (reads) \( x \). A group \( g_j \text{reads}_x \text{from } g_i \) if \( g_j \text{ reads a value of } x \text{ written by } g_i \).

Two groups \( g_i \) and \( g_j \) are said to **conflict** if there are two logically conflicting transactions \( t_i \in g_i \) and \( t_j \in g_j \).

### 5.3.2. Group Concurrency Control Protocols

We now describe a family of concurrency control protocols based on the concept of groups as "high level" transactions. In Section 5.4 we show that any member of this family ensures one-copy serializability. In Section 5.5, we show that several previously known protocols are members of this set. A group concurrency control protocol \( \pi \) partitions transactions into disjoint groups, and has the
following two components:

1. A Local Policy $P_g$, for each group $g$, ensures the one-copy serializability of
   (all transactions in) $g$. The local order $<_g$ is defined to be the serialization
   order of all transactions in $g$ given by $P_g$.

2. A Global Policy $P$ ensures a total order $<$ on all groups such that:
   if $g'$ reads $x$ from $g$, and $g \neq g'$, then
      a. $g \leq g'$.
      b. There is no $g''$ that writes $x$ such that $g < g'' < g'$.
      c. $g'$ reads the final value of $x$ in $g$, with respect to $<_g$, given by $P_g$.

The group ordering $<$ and the serialization ordering of transactions in each
group, define a total order $O$ of all transactions. The policies must ensure that this
total order is a correct serialization order, i.e., is equivalent to a serial one-copy
log. Informally, condition 2.a ensures that the order on groups is consistent with
the reads $x$ from relation. Conditions 2.b and 2.c ensure that in the total order $O$
no transaction $t_k$ that writes $x$ is ordered between two transactions $t_i$ and $t_j$
where
$t_j reads x from t_i$. In the next section we present a proof that $O$ is a correct one-
copy serialization order, i.e., group concurrency control protocols ensure one-copy
serializability.

We should note that although our concept of groups might superficially be
similar to the nested transaction structure [Moss82, Raeu86], they are actually
very different. In the nested transaction model, a transaction may invoke several
sub-transactions, which may themselves invoke other sub-transactions, thus form-
ing a tree structure. The nested transaction as a whole is synchronized with other
nested transactions at the root level, and the different sub-transactions in the same
nested transaction synchronize with one another. Hence the similarity with our
concept of a group, where transactions in a group synchronize together, and groups
as a whole synchronize at a higher level. However, this is where the similarity
ends. Most significantly, the group concept is a convenient model for characterizing
different concurrency control protocols, while the nested transaction construct
uses a specific concurrency control mechanism and a nested transaction is designed
in a structured manner as a single entity. The only relation between transactions
in the same group is that they use the same local policy, which might differ
between different groups. In nested transactions the same concurrency control
mechanism is used by all transactions, and a special mechanism is used to
correctly synchronize the accessing of data by different siblings in the tree struc-
ture.

5.4. Serialization Group Model

In this section we show that any group concurrency control protocol ensures
one-copy serializability. The local policies ensure that each group $g$ is one-copy
serializable with some order $<_g$. We now show that the conditions on the global
policy are sufficient to ensure one-copy serializability of all transactions.

Let $L$ be a log over a set of transactions $T$ that are partitioned into groups.
The local policy $P_g$ of group $g$ ensures that $g$ is one-copy serializable according to a
total order $<_g$. A graph $G$ is a one-copy serialization group graph $1-SGG[L]$ for
log $L$ with a local order $<_g$ for each group $g$ if $G$ satisfies the following conditions:

1. The nodes of $G$ represent the groups that partition $T$. 
2. If \( g_j \lex x \from g_i \), then \( G \) contains an edge from \( g_i \) to \( g_j \). This edge is called a group \( \lex x \from \) edge. If the value read by \( g_j \) is not the final value of \( x \) with respect to \( g_i \) and local order \( <_{g_i} \), then \( G \) must contain an edge from \( g_j \) to \( g_i \) (note that this immediately creates a cycle). This edge is called a group indirect \( \lex x \before \) edge.

3. For each object \( x \), \( G \) embodies a total order over all groups that write \( x \), i.e., for each pair of groups \( g \) and \( g' \) that write \( x \), there is either a path in \( G \) from \( g \) to \( g' \) or from \( g' \) to \( g \). This total order is called a group write order for \( x \), and is denoted \( \Rightarrow_{x}^{g} \).

4. For each object \( x \) and groups \( g_i, g_j, g_k \) such that \( g_j \lex x \from g_i \) and \( g_i \Rightarrow_{x}^{g} g_k \), \( G \) contains a path from \( g_j \) to \( g_k \). This path is called a group \( \lex x \before \) path.

Reads_from edges (henceforth, we will refer to group edges simply as reads_from, indirect reads_before, etc., unless an ambiguity may arise) capture the conflicts arising from reads_from relations between different groups; while indirect reads_before edges capture conflicts resulting from a group reading an object \( x \) from another group \( g \), where the value read is not the final value of \( x \) with respect to \( g \) with order \( <_{g} \). However, these edges are not enough to capture all group conflicts, and hence to establish the one-copy serializability of all transaction executions. Therefore \( 1-SG[Q] \) must also have enough edges to embody, for each object \( x \), a total order on all groups that write \( x \). Furthermore, \( 1-SG[Q] \) must contain all reads_before paths resulting from the write orders chosen.
Note that if each group contains one transaction only, then \(1 - SGG[L]\) degenerates to the logical serialization graph defined in [Bern86]. In [Bern86] it is proved that the existence of an acyclic logical serialization graph is a necessary and sufficient condition to ensure one-copy serializability. With our model, the existence of an acyclic \(1 - SGG[L]\) does not guarantee one-copy serializability. Each group is a set of transactions that may not be serializable. However, we can show that:

**Theorem 5.1:** Let \(L\) be a log over a set of transactions \(T\). \(L\) is one-copy serializable if and only if \(T\) can be partitioned into groups such that:

1. Each group \(g\) is one-copy serializable, with order \(<_g\).
2. There is a corresponding acyclic \(1 - SGG[L]\).

**Proof:** (If part). Assume for log \(L\) that \(T\) can be partitioned into groups such that each group \(g\) is one-copy serializable with some order \(<_g\), and \(1 - SGG[L]\) is acyclic. We now show how to induce a serial one-copy log \(L_s\) over \(T\), and then prove that \(L_s\) is equivalent to \(L\), thus we prove that \(L\) is one-copy serializable.

Since \(1 - SGG[L]\) is acyclic, we can topologically sort \(1 - SGG[L]\) and construct a serial one-copy log \(L_s\) as follows: (1) For each transaction \(t_i\), construct a serial one-copy transaction log by concatenating \(t_i\)'s logical operations in any order consistent with \(<_i\). (2) For each group \(g_i\), construct a serial one-copy group log over all transactions in \(g_i\) by concatenating the serial transaction logs in \(<_{g_i}\) order. (3) Construct \(L_s\) by concatenating the serial group logs according to the topological order. We now prove that \(L_s\) is equivalent to \(L\).
1. If \( t_j \) reads \( x \) from \( t_i \) in \( L \), we prove that \( t_j \) reads \( x \) from \( t_i \) in \( L_s \). Let \( t_k \) be any other transaction that writes \( x \), where \( t_i \in g_i, t_j \in g_j \) and \( t_k \in g_k \). We now prove that \( t_k \) cannot be ordered between \( t_i \) and \( t_j \) in \( L_s \).

If \( g_i = g_j = g \), then \( t_i \) and \( t_j \) execute in the same group \( g \), which from condition (1) of the theorem is one-copy serializable with some total order \( <_g \) on all transactions in \( g \). There are two cases to consider. First \( t_k \in g \). Since \( <_g \) is a one-copy serialization order for transactions executing in \( g \), and \( t_j \) reads \( x \) from \( t_i \), there is no other transaction \( t_k \in g \) that writes \( x \) such that \( t_i <_g t_k <_g t_j \). Hence, from step 2 in the construction of \( L_s \), there is no transaction \( t_k \in g \) that writes \( x \) and is ordered in \( L_s \) between \( t_i \) and \( t_j \). Second, consider the case where \( t_k \notin g \). From step 3 in the construction of \( L_s \), there is no transaction \( t_k \) member of another group that is ordered in \( L_s \) between \( t_i \) and \( t_j \).

If \( g_i \not\approx g_j \) then there are three cases to consider depending on the order between the two groups \( g_i \) and \( g_k \) that write \( x \) in \( \Rightarrow^g_\bar{\sim} \).

Case 1. \( g_k \not\Rightarrow^g \bar{\sim} g_i \). By definition, since both \( g_i \) and \( g_k \) write \( x \), \( 1-SGG[L] \) contains a path from \( g_k \) to \( g_i \). Therefore \( g_k \) is topologically sorted before \( g_i \), and hence \( t_k \) precedes \( t_i \) in \( L_s \).

Case 2. \( g_k = g_i \). Assume \( t_i <_g t_k \). Since \( t_j \) reads \( x \) from \( t_i \) and both \( t_i \) and \( t_k \) write \( x \), then \( g_j \) does not read the final value of \( x \) with respect to \( g_i \), and the local order \( <_{g_i} \). Hence, \( 1-SGG[L] \) must have an indirect reads..before edge from \( g_j \) to \( g_i \). But \( 1-SGG[L] \) also contains a reads..from edge from \( g_i \) to \( g_j \), therefore \( 1-SGG[L] \) has a cycle contradicting assumption that \( 1-SGG[L] \) is acyclic. Thus \( t_k <_g t_i \) and hence, \( t_k \) precedes \( t_i \) in \( L_s \).
Case 3. \( g_i \Rightarrow \frac{g}{g} g_k \). There are two cases to consider. First, suppose that \( g_k \neq g_j \). Since \( t_j \text{ reads}_x \text{ from } t_i \), then \( 1 - SGG[L] \) contains a reads-before path from \( g_j \) to \( g_k \). Therefore \( g_k \) is topologically sorted after \( g_j \), and hence \( t_k \) must follow \( t_j \) in \( L_s \). Now suppose that \( g_k = g_j \). Since \( t_j \text{ reads}_x \text{ from } t_i^{g_j} \), and \( t_i^{g_j} < t_k \) (and \( t_k \) writes \( x \)), then \( t_j < t_k \). Hence, \( t_k \) must follow \( t_j \) in \( L_s \).

Hence, in all three cases, \( t_k \) cannot be ordered between \( t_i \) and \( t_j \) in \( L_s \), thus \( t_j \text{ reads}_x \text{ from } t_i \) in \( L_s \).

2. If \( t_j \text{ reads}_x \text{ from } t_i \) in \( L_s \), we prove that \( t_j \text{ reads}_x \text{ from } t_i \) in \( L \). By definition \( t_j \text{ reads}_x \text{ from } t \) for some transaction \( t \) in \( L \). From the part (1) of the proof, \( t_j \text{ reads}_x \text{ from } t \) in \( L_s \). Since \( \text{reads}_x \text{ from} \) relations are unique, i.e., a transaction can \( \text{reads}_x \text{ from} \) one transaction only, \( t \) is \( t_i \), and so \( t_j \text{ reads}_x \text{ from } t_i \) in \( L \).

(Only if part). Assume that \( \log L \) over a set of transactions \( T \), is equivalent to a serial one-copy \( \log L_s \). We now show how to partition \( T \) into groups and construct a corresponding acyclic one-copy serialization group graph for \( L \).

Let each transaction form a group by itself, i.e., each transaction \( t_i \) forms a group \( g_i \). Let \( G \) be a graph whose nodes are groups and whose edges are: \( \{ g_i, g_j \} \mid t_i \text{ precedes } t_j \text{ in } L_s \}. \) Since each group contains one transaction only, then trivially all groups are one-copy serializable. Furthermore, since \( L_s \) induces a total order on all transaction executions, \( G \) is acyclic. We now prove that \( G \) is a \( 1-SGG[L] \) by showing that \( G \) satisfies the requirements of a one-copy serialization group graph.
1. The nodes of $G$ represent the groups that partition $T$.

2. **Reads-from edges.** If $g_j \text{ reads } x \text{ from } g_i$, then $t_j \text{ reads } x \text{ from } t_i$ in $L$. Since $L$ is equivalent to $L_s$, then $t_j \text{ reads } x \text{ from } t_i$ in $L_s$. Since $t_i$ precedes $t_j$ in $L_s$, <$g_i, g_j>$ is an edge in $G$.

3. **Indirect reads-before edges.** Each group contains only one transaction, hence, if $g_j \text{ reads } x \text{ from } g_i$ in $L$, then $g_j$ can only read the final value of $x$ with respect to $g_i$ and the local order <$_{g_i}$. Thus $G$ has no indirect reads-before edges.

4. **Write order.** $L_s$ totally orders all transactions, and therefore embodies, for each object $x$, a total order $\Rightarrow_x$ on all transactions that write $x$, such that if $t_i$ and $t_j$ write $x$, and $t_i$ precedes $t_j$ in $L_s$, then $t_i \Rightarrow_x t_j$. Hence, $G$ embodies, for each object $x$, a total order $\Rightarrow x$ on all groups that write $x$, where $g_i \Rightarrow x g_j$ if and only if $t_i \Rightarrow x t_j$.

5. **Reads-before paths.** If $g_j \text{ reads } x \text{ from } g_i$ and $g_i \Rightarrow x g_k$, we show that $G$ has an edge from $g_j$ to $g_k$. Since $g_j \text{ reads } x \text{ from } g_i$, then $t_j \text{ reads } x \text{ from } t_i$ in both $L$ and $L_s$. Since $g_i \Rightarrow x g_k$, then $t_i \Rightarrow x t_k$. But $L_s$ is a serial one-copy log, hence $t_j$ must precede $t_k$, and therefore $G$ has an edge from $g_j$ to $g_k$.

Thus, $G$ is an acyclic $1-SGG[L]$ with respect to the set of groups \{\{g_i\mid g_i \text{ contains only } t_i\}\}.

We now show that group concurrency control protocols ensure conditions 1 and 2 of Theorem 5.1, and hence:

**Theorem 5.2:** A group concurrency control protocol ensures one-copy serializability.
Proof: Consider a group concurrency control protocol $\pi$. Let $L$ be a log resulting from the execution of a set of transactions $T$ using this protocol. $\pi$ partitions $T$ into disjoint groups $V = \{g_1, g_2, \ldots, g_k\}$. Let $P_g$ denote the local policy of $\pi$ for group $g$, and $P$ denote the global policy of $\pi$. Let $G$ be a graph $\langle V, E \rangle$ where $E$ is $\{<g_i, g_j> | g_i < g_j, \text{where} < \text{is the group order ensured by} P\}$. To prove that $\pi$ ensures one-copy serializability, we prove that conditions (1) and (2) of Theorem 5 are satisfied: we show that (1) each group $g$ is one-copy serializable with some order $<_g$, (2) $G$ is a $1-\text{SGG}[L]$, and (3) $G$ is acyclic. By definition, each local policy $P_g$ ensures that each group is one-copy serializable, with order $<_g$. To prove that $G$ is an $1-\text{SGG}[L]$, we have to show that $G$ satisfies the requirements of one-copy serialization graphs.

1. By definition, the nodes of $G$ represent the groups that partition $T$.

2. Reads_from edges. If $g_j \text{reads}_x \text{from} g_i$, then condition 2.a of the global policy $P$ guarantees that $g_i < g_j$. Hence, $<g_i, g_j>$ is an edge in $G$.

3. Indirect reads_before edges. The global policy $P$, condition 2.c, ensures that if $g_j \text{reads}_x \text{from} g_i$, then $g_j$ reads the final value of $x$ with respect to $g_j$. Hence, $G$ has no indirect reads_before edges.

4. Write order. The group order $<$ totally orders all groups, hence, $G$ embodies a total order on all groups. Therefore, $G$ embodies for each object $x$, a total order $\Rightarrow^g_x$ on all groups that write $x$, and is defined as follows: if $g_i$ and $g_j$ write $x$ such that $g_i < g_j$, then $g_i \Rightarrow^g_x g_j$.

5. Reads_before paths. Suppose that $g_j \text{reads}_x \text{from} g_i$, and $g_i \Rightarrow^g_x g_k$. Since $g_j \text{reads}_x \text{from} g_i$, then from condition 2.b of the global policy there is no $g_k$
that writes $x$ such that $g_i < g_k < g_j$. Since $<$ totally orders all groups, then $g_k < g_i$ or $g_j < g_k$. Assume $g_k < g_i$. Since both $g_k$ and $g_i$ write $x$, then by definition $g_k \Rightarrow \lnot x g_i$ contradicting assumption that $g_i \Rightarrow \lnot x g_k$. Hence $g_j < g_k$, and $<g_j,g_k>$ is an edge in $G$, i.e., $G$ contains a reads-before path from $g_j$ to $g_k$.

Thus $G$ is a $1-SGG[L]$. Finally, $G$ is acyclic since $<$ is a total order on all groups and the edges of $G$ are defined according to $<$. Hence, for a log $L$ over transactions $T$, $\pi$ partitions $T$ into groups such that (1) all groups $g$ are one-copy serializable according to some order $<_g$, and (2) log $L$ has a corresponding acyclic $1-SGG[L]$. Therefore, from Theorem 5.1, $\pi$ ensures that log $L$ is one-copy serializable. $\Box$

5.5. Analysis of Protocols

We will show that several known concurrency control protocols [Giff79, Eage83, Skee84, Herl87] are instances of our group paradigm. In this section we show that they all partition transactions into groups, where each group uses a local policy to ensure the one-copy serializability of all transactions in that group. Furthermore, each protocol uses a global policy that ensures a group order on all groups, and hence, from Theorem 5.2 ensures one-copy serializability.

Little work has been done comparing the level of data availability attained by concurrency control protocols or the costs they incur in a database that may suffer from site and communication failures. Coan et al. [Coan86] analyze the availability level attained by different protocols assuming (1) that each object is fully replicated, that is, each object has a copy on every site, and (2) that partitioning failures are limited to the single case where the network is divided into two
partitions only. Using these strong assumptions, a maximum availability level is
defined, and it is shown that most of the protocols discussed, including the virtual
partitions and the accessibility thresholds protocols, attain this maximum level of
availability (no analysis is made of the costs incurred by the protocols). In this
section we compare concurrency control protocols in the way they achieve correct-
ness, and we make no assumptions about object replication or about how failures
may partition the database.

We start our analysis with Gifford's quorum protocol [Giff79], where all tran-
sactions are members of one group only. This is followed by the missing write pro-
tocol [Eage83], where transactions use two different local policies, and the class
conflict protocols [Skee84], which only handles the case where the network parti-
tions (and no recovery takes place). We then show that both the virtual partitions
protocol, and the accessibility threshold protocol are instances of the paradigm.
The proof of correctness for the accessibility threshold protocol is much simpler
than the proof presented in Chapter 4. Furthermore, the proof illustrates the
major mechanisms used by the protocol to ensure correctness. Finally, we prove
that Herlihy's quorum consensus protocol [Herl87] is also an instance of the group
paradigm.

5.5.1. Gifford's Quorum Protocol

The quorum protocol [Giff79] presented by Gifford, maintains correctness by
requiring all logically conflicting operations to physically conflict. This protocol
represents a degenerate case of the paradigm, where all transactions are members
of the same group, and use the same local policy. This policy associates with each
object \( x \) two integers: the read quorum, \( q_r[x] \), and the write quorum, \( q_w[x] \). The read and write quorums determine the number of copies accessed or written by a logical read or write operation. Version numbers are used as described in Chapter 2, and all transactions use a conflict preserving concurrency control protocol. The local policy requires each set of size \( q_w[x] \) to have at least one copy in common with any set of size \( q_r[x] \), or \( q_w[x] \). (Thomas [Thom79] describes a special case of this protocol, where read and write operations are required to access and write a majority of copies.)

Since all transactions use the same local policy, transactions form one group only, and the conditions on the global policy are trivially satisfied. It is also straightforward to prove that the local policy ensures that the group is one-copy serializable. Informally, for each log \( L \), construct an \( SG[L] \). From the restrictions on quorums, any two logically conflicting operations must also physically conflict. Hence \( 1 - SG[L] \) is the same as \( SG[L] \). But all transactions use a conflict preserving concurrency control protocol, hence \( SG[L] (\equiv 1 - SG[L]) \) is acyclic, and therefore the group is one-copy serializable. A similar, but more detailed proof is given in [Bern83].

5.5.2. The Missing Write Protocol

In this section, we present a simplified version of the missing write protocol [Eage83] presented by Eager and Sevcik. This protocol is a special instance of the group paradigm where transactions use two kinds of local policies: the normal mode policy, and the failure mode policy. The normal mode policy executes transactions using a read-one, write-all approach, i.e., read quorums are one copy, and
write quorums are all copies of the object. The failure mode policy executes transactions using read and write quorums satisfying the Gifford quorum conditions. Both policies require transactions to use a conflict preserving concurrency control protocol. A normal (failure) group is defined as all transactions using the normal (failure) mode policy. Hence, as we proved in the previous section, each group is one-copy serializable.

The global policy imposes a group order $<$ on these two groups such that the normal group precedes the failure group. The global policy is enforced as follows. A transaction $t_i$ starts using the normal mode policy, and if it cannot execute a physical write operation on a copy of object $x$ (due to site or partitioning failures), it switches to the failure mode policy. Assume that $t_i$ was writing object $x$. The protocol requires that $t_i$ leave a missing write token for $x$ at all copies that it accesses or writes. When a normal mode transaction encounters any missing write token, it also switches to failure mode and leaves missing write tokens for all copies that it is aware of at all copies it accesses or writes.

Now we show that the global policy ensures conditions 2.a, 2.b, and 2.c. We only have to consider the cases when a transaction in the failure group reads from a transaction in normal group (later we will briefly discuss how recovery from failures takes place).

(a) Any transaction in the failure group $g_f$ either reads the value written by another transaction in the same group, or in the normal group $g_n$. But, by definition, the normal group precedes the failure group. Hence, if $g_f$ reads $x$ from $g_n$, then $g_n < g_f$. 
(b) If $g_f$ reads $x$ from $g_n$, then by definition, there is no third group that is ordered between the normal and the failure groups.

(c) Finally, let $g_f$ reads $x$ from $g_n$. Any failure group transaction $t_f$ leaves missing write tokens at a read quorum of $x$. Any normal group transaction writes all copies of $x$, and hence must encounter at least one such token, and hence, will switch to the failure mode policy. Therefore, once $t_f$ is executed, no other normal group transactions are executed. Furthermore, $t_f$ reads the value with the highest version number, and hence, it reads the final value of $x$ with respect to the local order $<_g$ on all transactions in the normal group.

Once sites and communication links recover, a special copier transaction is executed at all sites that performs the following: (1) For each copy $x_s$ that missed some write operations, the copier transaction accesses a failure mode read quorum of copies and writes $x_s$ with the value of the copy with the highest version number. (2) After $x_s$ is updated, the missing write tokens of $x_s$ are deleted from all sites. (In [Eage83], some of the updates can be executed during the execution of the user transactions to increase efficiency and to avoid unnecessary execution of transactions using the failure mode policy.) Hence, subsequent to failures and recovery, the copier transactions "logically" reinitialize the database to ensure that all transactions use the normal mode policy, and hence, form a new normal group. Therefore, groups alternate in pairs between normal and failure.

5.5.3. Class Conflict Protocol

In the class conflict protocol proposed by Wright and Skeen [Skee84, Wrig83], transactions are a priori divided into classes [Bern80], which may be well defined
transaction types or may be syntactically defined by the objects a transaction may read or write. It is assumed that failures can be detected correctly whenever they occur, and when a partitioning failure occurs, each partition must decide which classes of transactions it will allow to execute.

All transactions executing in a partition form a group, with all transactions executing in the initial configuration forming a group \( g_0 \). Transactions executing in a partition use any correct concurrency control protocol, and a read-one write-all (copies in partition) protocol. Hence, each group is one-copy serializable.

The global policy of this protocol uses class conflict graphs to determine which transactions to allow to execute in each partition. This graph corresponds to a \( 1-SGG[L] \), where nodes are classes of transactions, and edges correspond to potential logical conflicts between classes. Unlike the \( 1-SGG[L] \), which is used for proof purposes only, in the class conflict protocol each partition constructs a class conflict graph containing all classes that it may execute as well as classes that may be executed in other partitions.

Each partition analyzes its class conflict graph, and checks whether the graph contains any cycles that span more than one partition (called multi-partition cycles). If any such cycles exist, the partition must delete enough classes until the graph contains no multi-partition cycles.

As presented, this protocol does not discuss what actions should be taken after recovery from partitioning. Hence, the only case handled is when the system is initially connected and then failures occur that lead to partitioning. The class conflict graph ensures a total order on all classes executing in different partitions,
hence, a total order $<$ on all groups can be defined to be consistent with that order, with group $g_0$ preceding all other groups. We now show that the global policy satisfies conditions 2.a, 2.b, and 2.c of the group paradigm:

(a) Transactions in $g_0$ only read values written in $g_0$. Any transaction $t_j$ in group $g_j$ reading object $x$, either reads the value written by another transaction in the same group $g_j$ or in $g_0$, which precedes $g_j$. Hence, if $g_j \text{ reads } x \text{ from } g_i$, it must be the case that $i = 0$, and therefore $g_i < g_j$.

(b) Let $g_j \text{ reads } x \text{ from } g_0$. Assume that $g_k$ writes $x$, where $g_0 < g_k < g_j$. Since $g_k < g_j$, then the class conflict graph contains a path from $g_k$ to $g_j$. Furthermore, since $g_k$ writes $x$, and $g_j$ reads $x$, then any class conflict graph contains an edge from $g_j$ to $g_k$ (this edge is the result of a potential conflict between a transaction in $g_j$ that reads object $x$, which may be written by a transaction executed in $g_k$). Hence, the class conflict graph contains a cycle, contradicting the protocol, which ensures no multi-partition cycles. Therefore, if $g_j \text{ reads } x \text{ from } g_0$, then there can be no $g_k$ that writes $x$, where $g_0 < g_k < g_i$.

(c) Let $g_j \text{ reads } x \text{ from } g_0$. Every write operation in $g_0$ writes all copies of $x$; hence, if $g_j$ reads $x$, it must read the final value of $x$ in $g_0$ with respect to $<_{g_0}$.

5.5.4. The Virtual Partitions Protocol

We now show that the virtual partitions protocol is an instance of the group paradigm. A set of sites that can communicate together form a virtual partition (where all sites in the virtual partition have the same view and view-id). Transac-
tions executing in the same virtual partition form a group. The local policy executes a read $x$ by accessing any copy of $x$ in the virtual partition; and executes a write $x$ by writing all copies of $x$ that reside on sites in the virtual partition. Furthermore, all operations are restricted to execute on sites in the same virtual partition, i.e., sites that have views with the same view-id. Hence, any two logically conflicting operations executed in the same group must also physically conflict. Since all transactions use a conflict preserving concurrency control protocol, each group is one-copy serializable.

The global policy uses the following mechanisms.

1. **The Accessibility Rule.** An object $x$ is accessible in a virtual partition only if a majority of copies of $x$ reside on sites in that partition. A transaction may read or write only those objects that are accessible in its virtual partition.

2. **The View Management Protocol.** The view management protocol is a two phase protocol that creates virtual partitions and updates views. The protocol ensures that all virtual partitions are totally ordered according to some total order $<<$ that satisfies the following condition:

   Let $vp_1$ be a virtual partition whose sites have view $v_1$, and $vp_2$ be a virtual partition whose sites have view $v_2$. Let $s$ be a member of virtual partition $vp_1$ and view $v_2$. If $vp_1 << vp_2$ then $s$ must depart from $vp_1$ before any site $p$ can join $vp_2$.

3. **The Recovery Transaction.** After a site $s$ joins a new virtual partition $vp$, no read or write operation is allowed to access or write a copy residing on $s$ before a recovery transaction is executed. The recovery transaction brings the
copies of all accessible objects up-to-date. For each accessible object \( x \) in \( vp \), the transaction reads all copies residing on sites in \( vp \), and writes the value with the highest version number to all copies of \( x \) in \( vp \).

The view management protocol ensures a total order \( << \) on all virtual partitions. We define the total order \( < \) on all groups as follows. Let \( g_i \) be the set of all transactions executing in virtual partition \( vp_i \), and \( g_j \) the set of all transactions executing in virtual partition \( vp_j \). If \( vp_i << vp_j \), then \( g_i < g_j \). Each group \( g_i \) is uniquely identified by a virtual partition \( vp_i \), its view \( v_i \) and view-id \( v_i-id \). The next two lemma complete the proof that the virtual partitions protocol is a group concurrency control protocol.

**Lemma 5.3:** If \( g_j \) reads \( x \) from \( g_i \), then \( g_i < g_j \).

**Proof:** A user transaction always reads a value written by either another user transaction or by a recovery transaction in the same group. Hence, we only have to consider the case of a recovery transaction \( t_r \) in \( g_j \) reading a value written by a transaction in group \( g_i \). Let \( g_j \) be is the set of transactions executing in \( vp_j \). Note that \( t_r \) only accesses copies residing on sites in \( vp_j \). Since \( t_r \) accesses some copy \( x_s \) that was written by a transaction in group \( g_i \) (i.e., executing in virtual partition \( vp_i \)), then site \( s \) must have been a member of \( vp_i \) before being a member of \( vp_j \). For contradiction assume \( vp_j << vp_i \). From the definition of the view management protocol, if \( vp_j << vp_i \), then \( s \) must depart from \( vp_j \) before joining \( vp_i \). Hence, \( s \) is a member of \( vp_j \) before being a member of \( vp_i \); a contradiction. Therefore, \( vp_i << vp_j \) and thus \( g_i < g_j \). \( \square \)
Lemma 5.4: If $g_j$ reads $x$ from $g_i$, then:
1. there is no $g_k$ that writes $x$ such that $g_i < g_k < g_j$.
2. $g_j$ reads the final value of $x$ in $g_i$ with respect to local order $<_g_i$.

Proof: As in the previous lemma, we only consider the case of a recovery transaction $t_r$ executed in $g_j$, where $g_j$ is the set of transactions executing in virtual partition $vp_j$. For each accessible object $x$, $t_r$ accesses a majority of copies of $x$. Assume $w_k[x]$ is a write operation executed in virtual partition $vp_k << vp_j$. Every write operation must write at least a majority of copies, hence, $w_k[x]$ must write at least one copy that is accessed by $t_r$. All copies accessed by $t_r$ reside on sites in $vp_j$, and all copies written by $w_k[x]$ reside on sites in $vp_k$. From the view management protocol if $vp_k$ is a virtual partition where $vp_k << vp_j$, then any site $s$ that is in $vp_k$ must depart from $vp_k$ before joining $vp_j$. Hence, for all write operations $w_k[x]$ executed in $vp_k$, there is a copy that $t_r$ accesses after $w_k[x]$ writes. But $t_r$ reads the value of the copy with the highest version number, hence, if $g_j$ reads $x$ from $g_i$, then there is no $g_k$ that writes $x$ such that $g_i < g_k < g_j$, and the value read by $g_j$ is the final value of $x$ in $g_i$ with respect to local order $<_g_i$.

5.5.5. The Accessibility Threshold Protocol

The accessibility thresholds protocol presented in Chapter 4 is an instance of the group paradigm, where all transactions executing in the same view form a group. The local policies ensure the one-copy serializability of each group. In each view, an object $x$ is assigned read and write quorums $q_r(x,v)$ and $q_w(x,v)$ that satisfy Gifford's conditions on quorums: for an object $x$, a set of size $q_w(x,v)$ intersects with any set of size $q_w(x,v)$ or $q_r(x,v)$. A transaction executing in view $v$
executes a read of $x$ by accessing $q_r[x,v]$, and executes a write of $x$ by writing $q_w[x,v]$. Furthermore, a transaction executing in view $v$ is restricted to access or write copies that reside on sites with view $v$. These conditions, in addition to a conflict preserving concurrency control protocol ensure that each group is one-copy serializable.

The unique view-ids associated with views form a total order, thus defining a total order $<$ on all groups. Hence, each group is uniquely identified by a view and a view-id. The global policy uses the following two mechanisms.

1. **The Accessibility Threshold Restrictions.** Each object $x$ is assigned read and write accessibility thresholds $A_r[x]$ and $A_w[x]$. The sum of the read and write accessibility thresholds for each object $x$, i.e., $A_r[x] + A_w[x]$, must exceed the number of copies of $x$. Furthermore, write quorums of an object $x$ must always be greater than the write threshold for object $x$, i.e., $q_w[x,v] > A_w[x]$ for all views $v$. An object is read (write) accessible in a view $v$ if $A_r[x]$ ($A_w[x]$) copies of $x$ reside on sites in $v$. A transaction executing in view $v$ may only read (write) an object if it is read (write) accessible in $v$.

2. **The Update Transaction.** A site $s$ changes its view to $v$ (with view-id $v_{-id}$) by executing an update transaction. For all read accessible objects $x$ in $v$ with copies residing on $s$, the update transaction accesses $A_r[x]$ copies of $x$, reads the value with the highest version number, and updates the local copy $x_s$ with that value. An access operation is rejected by any site with a view-id greater than $v_{-id}$. If all access operations are successful, view $v$ is installed at $s$. When a site is accessed by an update transaction installing $v$,
it inherits \( v \), and installs view \( v \) or a view with a higher view_id.

For the purposes of the proof, an update transaction \( t \), executed during the installation of view \( v_j \), is a member of group \( g_j \), the set of all user transactions executing in \( v_j \). Informally, we note that since user transactions are restricted to read copies residing on sites in the same view, update transactions are the only transactions allowed to access copies written by transactions in a different group. An update transaction \( t \) installing a view \( v_j \) with view-id \( v_j\_id \) is restricted to read copies residing on sites whose views have view-ids less than or equal to \( v_j\_id \), thus ensuring global policy condition 2.a. Furthermore, the execution of an update transaction, and the restrictions on accessibility thresholds and write quorums ensure that once update transaction \( t \) (in group \( g_j \)) reads an object \( x \), written by group \( g_i \), no write operation is allowed to be executed in a group \( g_k \) with a view-id less than \( v_j\_id \) (i.e., any group \( g_k < g_j \)). Since a read operation reads the copy with the highest version number of \( A_r[x] \) copies and each write \( x \) writes at least \( A_w[x] \), the value read by \( t \) is the most up-to-date value, i.e., if \( g_j \) reads \( x \), which was written in \( g_i \), then (1) there is no \( g_k \) that writes \( x \) such that \( g_i < g_k < g_j \), and (2) \( g_j \) reads the final value of \( x \) in \( g_i \) with respect to local order \(<_{g_i}\), thus satisfying conditions 2.b and 2.c of the global policy. We now formally prove that the global policy satisfies conditions 2.a, 2.b, 2.c of the group paradigm, and hence, the accessibility thresholds protocol is a group concurrency control protocol.

**Lemma 5.5:** If \( g_j \) reads \( x \) from \( g_i \), then \( g_i < g_j \).

**Proof:** Let view \( v_j \) be associated with group \( g_j \). A user transaction always reads a value written by either a user or an update transaction executing in the
same group. Hence, the only case where \( g_i \) and \( g_j \) are distinct groups is when an update transaction in \( g_j \) is installing view \( v_j \). But an access operation executed by an update transaction is rejected by any site with a view whose view-id is greater than the view-id of \( v_j \). Therefore if an update transaction is terminated successfully, it either read a copy written by a transaction in group \( g_j \) or in another group \( g_i \) where \( g_i < g_j \). □

**Lemma 5.6:** If \( g_j \) reads \( x \) from \( g_i \), then:

1. there is no \( g_k \) that writes \( x \) such that \( g_i < g_k < g_j \).
2. \( g_j \) reads the final value of \( x \) in \( g_i \) with respect to local order \( <_{g_i} \).

**Proof:** As in the previous lemma, we only consider the case of an update transaction \( t_j \) in \( g_j \) that installs view \( v_j \) with view-id \( v_j-id \). For read accessible object \( x \), \( t_j \) accesses \( A_r[x] \) copies of \( x \). Let \( w_k[x] \) be any write operation executed in a group \( g_k \), where \( g_k < g_j \). Since \( w_k[x] \) writes at least \( q_w[x,v] > A_w[x] \) copies of \( x \), there must be at least one copy that is accessed by \( t_j \) and written by \( w_k[x] \). All \( A_r[x] \) sites accessed by \( t_j \) either install \( v_j \) or a view with a higher view-id. Furthermore, all copies written by \( w_k[x] \) must reside on sites with view \( v_k \). Hence, once the update transaction \( t_j \) is executed, no write operations can be executed in group \( g_k \) where \( g_k < g_j \).

Therefore, for all write operations \( w_k[x] \) executed in \( g_k \), there is a copy that \( t_j \) accesses after \( w_k[x] \) writes. But \( t_j \) reads the value of the copy with the highest version number. Hence, if \( g_j \) reads \( x \) from \( g_i \), then there is no \( g_k \) that writes \( x \) such that \( g_i < g_k < g_j \) and the value read by \( g_j \) is the final value of \( x \) in \( g_i \) with respect to local order \( <_{g_i} \). □
5.5.6. Herlihy's Quorum Consensus Protocol

Herlihy [Herl87] presents a generalization of Gifford's quorum protocol for multi-version databases, where a transaction chooses a natural number, called its level, to execute in. Each object $x$ has a quorum assignment table, which assigns for each level $l$, a read quorum $q_r[x,l]$ and a write quorum $q_w[x,l]$. Initially, each level has an original quorum assignment binding with an initial timestamp. Subsequently, the quorum assignment bindings may be changed, in which case the timestamp associated with the level is updated to reflect the new binding. Each copy stores a sequence of timestamped versions, one for each level containing the most recently written version, if any. Each transaction is assigned a unique timestamp that determines its serialization order. A transaction $t$ executing in level $l$ writes an object $x$ by writing the $l^{th}$ version of $q_w[x,l]$ copies of $x$. Each version written is assigned $t$'s timestamp. A transaction $t$ executing in level $l$ reads an object $x$ by accessing $q_r[x,l]$ copies of $x$. For each accessed copy $t$ accesses the version associated with the highest level less than or equal to $l$, and reads the copy with the highest timestamp.

This protocol fits neatly into our paradigm. All transactions executing at the same level, and that use the same quorum assignment binding form a group. All groups use a local policy that synchronizes transactions by a mechanism capable of assigning logical timestamps [Lamp78] to transactions so that their timestamp ordering reflects their serialization order [Reed83]. Hence, each group is one-copy serializable.
A group is uniquely identified by the pair \( <l,ts> \), where \( l \) is the level transactions execute in, and \( ts \) is the timestamp of the quorum binding. By definition, both levels and timestamps are totally ordered, hence, there is a group order \( < \) on the set of all groups as follows. For \( g = <l,ts> \) and \( g' = <l',ts'> \), \( g < g' \) if \( l < l' \) or if \( l = l' \) and \( ts < ts' \). The global policy uses the following mechanisms.

1. The Ratchet Lock and Quorum Assignment Bindings. Each copy has a ratchet lock, which is a counter that records the highest level of a transaction that has read that copy. A copy rejects write requests from any transaction whose level is less than the copy’s ratchet lock. Writes are said to be enabled for an object at level \( n \) if none of its ratchet locks exceed \( n \). Furthermore, a copy with quorum assignment binding \( ts \) for level \( n \) rejects all operations executed at level \( n \) but with quorum assignment bindings \( ts' < ts \).

2. The Quorum Intersection Invariant: Quorum assignments are required to satisfy the quorum intersection invariant. If writes are enabled to object \( x \) at level \( l \), then any set of size \( q_w[x,l] \) must have at least one copy in common with any set of size \( q_r[x,m] \), for all levels \( m \geq l \).

3. The Deflation Transaction. A copy changes the quorum assignment binding of object \( x \) for level \( l \) from \( q_r[x,l] \) and \( q_w[x,l] \) (called old read and write quorums) to \( q_r[x,l'] \) and \( q_w[x,l'] \) (called new read and write quorums), where \( l' \) is the same level as \( l \) but with the new quorum assignment binding, by atomically executing a deflation transaction. The deflation transaction:

   (1) Accesses \( q_r[x,l] \) copies of \( x \), and reads the closest preceding version for level \( l \) with the highest timestamp.

   (2) Disables write operations at all levels \( m < l \) with write quorums
$q_w[x,m]$, where a set of size $q_w[x,m]$ does not intersect with a set of size $q_r[x,l']$. This is done by advancing the copies' ratchet locks beyond $m$.

(3) Writes the read version and its timestamp to $q_w[x,l']$ copies of $x$.

(4) Writes the highest ratchet lock to $q_r[x,l']$ copies of $x$.

(5) Updates the quorum assignment tables and their associated timestamp bindings, at enough copies (called read and write coquorum), so that these updated copies have a nonempty intersection with every set of size $q_r[x,l]$ and $q_w[x,l]$.

Informally, a read operation executed in group $g_j$ (where $g_j = \langle l_j, ts_j \rangle$) always reads the value of an object $x$ written in $g_j$ or in a group $g_i$ that precedes $g_j$ in group order $<$, thus satisfying condition 2.a of the global policy. The ratchet locks and timestamp binding, in addition to the quorum invariant, ensure that once $g_j$ reads $x$, no write operation is allowed to execute in any group $g_k$ that precedes $g_j$ in $<$. Since $g_j$ reads the the closest preceding version for level $l_j$, there is no $g_k$ that writes $x$ such that $g_i < g_k < g_j$, and the value read by $g_j$ is the final value of $x$ in $g_i$, thus satisfying conditions 2.b and 2.c of the global policy. Finally, the deflation transaction ensures that the quorum intersection invariant holds even when some quorum bindings are changed.

We note the similarities between the methods used by the global and local policies. In the local policy, which is essentially a timestamp-based protocol [Reed83], the timestamp order imposes a total order on all transactions in a group; in the global policy the pair $\langle$ level, timestamp $\rangle$ imposes a total order all groups. In the local policy, a read of $x$ reads the version written by the closest preceding
transaction in timestamp order; in the global policy, a read of \( x \) reads the version written by the closest preceding group in the group order. Finally in the local policy, once a read operation is executed, a read-timestamp disallows write operations with lower timestamps; in the global policy, once a read operation is executed, a ratchet lock disallows write operations from executing in preceding groups in the group order.

We now formally prove that the global policy satisfies conditions 2.a, 2.b, 2.c of the group paradigm. For the purposes of the proof, if deflation transaction \( t_d \) \texttt{reads\_x\_from} \( t \), where \( t \) is a member of \( g \), then \( t_d \) is considered a member of \( g \). Lemma 5.7 proves that if \( g_j \texttt{reads\_x\_from} \ g_i \) then \( g_i \) must precede \( g_j \) in the group order \(<\). Lemma 5.8 proves that if \( g_j \texttt{reads\_x\_from} \ g_i \) then (1) there is no other group that writes \( x \) and that is ordered between \( g_i \) and \( g_j \) in \(<\), and (2) the value read by \( g_j \) is the final value of \( x \) in \( g_i \) with respect to \(<_{g_i}\), and hence, the quorum consensus protocol is a group concurrency control protocol.

**Lemma 5.7:** If \( g_j \texttt{reads\_x\_from} \ g_i \), then \( g_i < g_j \).

**Proof:** Let \( g_j = <l_j,ts_j> \). By definition of read operations, \( g_i \) must have executed in a level less than or equal to \( l_j \). Furthermore, a read operation executed at level \( l_j \) with quorum assignment binding \( ts_j \) is rejected by any copy with quorum assignment binding \( ts > ts_j \). Hence, \( g_i < g_j \). \( \square \)

**Lemma 5.8:** If \( g_j \texttt{reads\_x\_from} \ g_i \), then:

1. there is no \( g_k \) that writes \( x \) such that \( g_i < g_k < g_j \).
2. \( g_j \) reads the final value of \( x \) in \( g_i \) with respect to local order \(<_{g_i}\).
Proof: We first consider the case where \( g_j \) has an original quorum assignment binding, and then consider the case where \( g_j \) does not.

Case 1. Suppose \( g_j = \langle l_j, ts_j \rangle \) has an original quorum assignment binding. From the quorum intersection invariant, any write operation executed in a level \( l_k \), where \( l_k < l_j \) must access at least one copy that is accessed by any read operation executed in \( g_j \). All copies read by \( g_j \) increment their ratchet locks, thus disallowing all write operations at levels \( l_k < l_j \) with original quorum assignment bindings. In addition, even when a deflation transaction changes the quorum binding of a level \( l_k < l_j \), step (4) of the deflation transaction updates the ratchet locks of a new read quorum of copies to \( l_j \). This ensures that once \( g_j \) reads \( x \), no write of \( x \) may be executed in a group with level less than \( l_j \). Since \( g_j \) reads the closest preceding version for \( l_j \), hence, if \( g_j \) reads \( x \) from \( g_i \) then there is no \( g_k \) that writes \( x \) such that \( g_i < g_k < g_j \). Furthermore, the read operation reads the version with the highest timestamp of group \( g_i \). Hence, the value read is the final value of \( x \) in \( g_i \) with respect to local order \( <_{g_i} \).

Case 2. Assume that \( g_j \) does not have an original binding. Consider any write operation executed in \( g_k \), where \( g_k = \langle l_k, ts_k \rangle \) and \( g_k < g_j \).

If \( l_j = l_k \) and \( ts_k < ts_j \), then a simple induction using steps (1), (3) and (5) of the deflation transaction shows that no write operations are executed in \( g_k \) after \( l_j \) changes its quorum assignment binding.

If \( l_k < l_j \) and a set of size \( q_w[x, l_k] \) intersects with a set of size \( q_r[x, l_j] \), then, as in case (1) once the quorum assignment binding of \( l_j \) is changed, no write operations are executed in \( g_k \).
If $l_k < l_j$ and a set of size $q_u[x, l_k]$ does not intersect with a set of size $q_r[x, l_j]$, then step (2) of the deflation transaction sets ratchet locks to disallow the execution of write operations in $g_k$ once the quorum assignment binding of $l_j$ is changed.

Hence, in all three cases, no write operation is executed in group $g_k < g_j$ after $g_j$ reads $x$. Since $g_j$ reads the closest preceding version for level $l_j$, hence, if $g_j$ reads $x$ from $g_i$, then there is no $g_k$ that writes $x$ such that $g_i < g_k < g_j$. Furthermore, the read operation reads the version with the highest timestamp of group $g_i$. Hence, the value read is the final value of $x$ in $g_i$ with respect to local order $<_{g_i}$.
CHAPTER 6

Conclusion

In this thesis we first developed two new concurrency control protocols that tolerate site, communication and partitioning failures. In neither protocol is it ever necessary for a read operation to physically access more than one copy. This contrasts with the protocols in [Giff79, Eage83], where in order to tolerate failures each read is required to access more than one copy. The choice of accessibility thresholds and quorums in the accessibility thresholds protocol provides the database designer with a large degree of flexibility in determining the availability of objects and the costs of executing operations. Accessibility thresholds determine the read and write availability of each object. To increase the read availability of an object, the read accessibility threshold is decreased, and vice-versa. Quorums determine the costs of executing read and write operations for each object in each new view. Hence unlike previous protocols [Giff79, Eage83], the cost of executing operations on an object is separated from the read and write availability of that object.

We also presented a paradigm to describe and prove the correctness of concurrency control protocols. We have shown that the concept of groups is fundamental to several protocols, especially those that are designed to handle partitioning failures. In contrast to previous models for concurrency control [Bern87b, Herl86], which concentrated exclusively on the issue of correctness, our paradigm
provides a modular approach for analyzing protocols as well as proving them correct. Instead of serializing all transactions using one "policy", our paradigm divides the problem into two tasks: first serializing transactions in each group using a local policy, and then serializing all groups using a global policy. The paradigm also simplifies the understanding of several previous protocols [Giff79, Eage83, Skee84, Herl87] as well as those presented in Chapters 3 and 4, and shows the similarities between them. Showing that a protocol is an instance of the paradigm is sufficient to prove its correctness.

The group paradigm provides insights for the development of new concurrency control protocols. There are two types of concurrency control protocols for serializing transactions in non-replicated databases: dynamic and static protocols. A static protocol, such as the timestamp protocol [Reed83], statically assigns to each transaction a timestamp that determines the transaction's serialization order. In contrast, a dynamic protocol such as two-phase locking [Eswa76] imposes no a priori ordering on transactions. The protocol ensures that transactions executing conflicting operations are dynamically ordered during execution. The protocols we have presented in this thesis have local policies that use the dynamic approach to serialize transactions in each group, and a global policy that uses the static approach to serialize groups with respect to each other. In fact, the serialization of groups is achieved by an a priori assignment of a unique identifier (the view-id) to each group. The quorum consensus protocol [Herl87] uses the static approach to serialize transactions within each group, and also to serialize groups.
The choice of a static or a dynamic protocol to serialize transactions within each group is independent of the method chosen to serialize groups. Thus, one may develop new protocols with a global policy that uses the dynamic approach to serialize groups; that is, an approach where the ordering of groups is determined dynamically during system execution.

As we described in this thesis, one way to use the concept of groups is for fault-tolerance; after a network partitioning, groups may correspond to transactions executing in the network partitions. However, the concept of groups is also useful in other ways, as explained below. Recall that a group is a set of transactions that are locally serialized with respect to each other, and that require minimal synchronization and communication with transactions in other groups. Synchronization with other groups may be required only at the time a group is formed (for example, by executing an update transaction). Thus, groups are naturally suited to serialize transactions in a hierarchical network consisting of several communicating local area networks. In such networks, local communication in each local area network is much faster and cheaper than communication between sites in different local area networks. With such a hierarchical network, each group naturally corresponds to all transactions executing in each local area network. This mapping minimizes the communication costs required to serialize transactions. Global one-copy serialization requires local serialization of the transactions within each group; that is, within each local area network where communication is inexpensive. Furthermore, minimal synchronization is needed between groups; that is, between different local area networks where communication is
slower and more expensive.

There are several possible extensions to the results of this thesis. We could extend the concept of groups from a 2-level hierarchy (a group as a set of transactions) into a 3-level hierarchy (a meta-group is a set of groups, which is a set of transactions), and in general to an n-level hierarchy. This would be naturally suited to serializing transactions in a large multi-level hierarchical network (where sites are connected by a local area network, clusters of local area networks are connected, clusters of clusters, etc.)

Another extension would be to implement the protocols presented in this thesis and to compare their performance with other concurrency control protocols in databases where partitioning failures may occur. As discussed in Chapter 4, our approach provides the database designer with several parameters to control data availability and the cost of operations. Implementing our protocols would allow us to study the effects these parameters on system performance, and thus further our understanding of database design.

A final extension to our model is the applicability of the group paradigm to a database model that exploits semantic knowledge about abstract data type objects. In this thesis we assumed that objects support only read and write operations. There are several protocols that exploit the semantic knowledge of abstract data type objects to increase data availability and concurrency [Garc83, Herl86, Jose85, Rreu86, Schw84]. We believe that a similar approach can be used to extend our group paradigm to databases with objects that support operations more complex than simple reads and writes.
References


[Herl87] Herlihy, M., "Dynamic Quorum Adjustments for Partitioned Data," Transactions on Database Systems 12, 2 (June 87), 170-194.


[Reed83] Reed, D.P., "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems* 1, 1 (February 1983) 3-23.


