Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik

Dissertation

Database Transaction Management in Mobile Ad-Hoc Networks

Sebastian Obermeier

in partial fulfillment of the requirements for the degree of
doctor rerum naturalium
(Dr. rer. nat.)

Paderborn,
May 27th, 2008

Abstract

With growing interest in mobile ad-hoc networks and increasing capabilities regarding processing power and connectivity, an interesting and important challenge is to combine database technology with mobile devices. However, a transfer of traditional research results to mobile devices is hindered by problems like message loss, unpredictable disconnections of mobile devices, and network partitioning. When applying database technology that was designed for traditional fixed-wired networks in such mobile environments, the problem of long or even infinitely long blocking times for commonly accessed resources arises. For instance, a message that releases blocked resources may never reach its destination and leads to blocking of resources longer than intended. To handle this kind of resource blocking, numerous synchronization strategies that reduce the overall goals "atomicity" and "full serializability" have been developed. However, reducing "atomicity" and "full serializability" can lead to data inconsistencies.

In this thesis, we strike a new path to reduce protocol blocking and transaction blocking for distributed transactions in mobile ad-hoc networks without abandoning the goals "atomicity" and "full serializability". We propose three main contributions in combination with a Web service transaction model that is especially targeted on dynamic service invocation in mobile ad-hoc networks.

Firstly, we present a technique that uses a special non-blocking state, the "Adjourn state", to eliminate the need of setting up predefined participant time-outs for aborting a transaction before the atomic commit protocol starts. This Adjourn state allows a flexible reaction to network failures and makes renewed invocations of sub-transactions superfluous in many cases. Our experimental evaluation proves the Adjourn state's reduction of transaction blocking and the increasing transaction throughput in unreliable ad-hoc networks.

Secondly, we develop an atomic commit protocol called the "Cross Layer Commit Protocol" (CLCP), to eliminate a single source of failure for atomic commit protocols. CLCP employs all transaction participants as multiple coordinators and

thus increases the protocol availability. In contrast to existing solutions, CLCP is a decentralized protocol that lets participants determine the transaction's decision by their own knowledge, making messages that inform participants of the transaction's decision superfluous. While traditional atomic commit protocols operate solely on the application layer, the cross-layer design of CLCP allows CLCP to operate very energy efficient. We prove the correctness of CLCP, including the liveness and safety property that are crucial for atomic commit protocols, and experimentally compare CLCP with other atomic commit protocols in a mobile ad-hoc environment. We show that CLCP significantly reduces the average blocking duration and that CLCP's energy consumption is remarkably small.

Thirdly, we present a new technique for treating blocked data of transaction participants that wait for a coordinator's commit decision. Our technique, Bi-State-Termination, gives participants that have moved during transaction execution the possibility to continue transaction processing before they know the coordinator's decision on transaction commit. The key idea of our technique is to consider both possible outcomes (commit and abort) of unknown transaction decisions. In contrast, traditional transaction processing would prevent participants that have moved during the atomic commit protocol execution and therefore have not received the transaction's commit decision from using parts of their own data in concurrent transactions. In mobile networks, there is no guarantee that these moved participants will ever receive the transaction's decision. Therefore, traditional transaction processing would cause resources to be unusable for an infinitely long period of time. In contrast, Bi-State-Termination even allows further transactions to use these blocked resources. We present three implementations for Bi-State-Termination and compare them experimentally. Furthermore, we use the TPC-C benchmark that captures a real world scenario with a high transaction load to prove that Bi-State-Termination-enabled transaction processing allows committing significantly more transactions than traditional transaction processing.

Due to the harmful effects that long transaction blocking has on concurrent transactions, the use of transaction processing in mobile networks was considered unfeasible, and research was targeting concepts that weaken atomicity, serializability, and data consistency. However, combining our three main contributions, the Adjourn State, the Cross Layer Commit Protocol, and the Bi-State-Termination, we can significantly reduce transaction blocking such that the duration and risk of infinite transaction blocking is, in many cases, not harmful to concurrent transactions any longer. Therefore, our contributions make the use of transaction processing

feasible even in unreliable mobile ad-hoc networks. This means, that we can give transactional guarantees as atomicity, serializability, and data consistency, which, in consequence, means that there is no longer the need to weaken these important transactional guarantees in mobile ad-hoc networks.

To Uta and our daughter Isabel Marie.

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. Dr. Stefan Böttcher, for his constant guidance and support in the process of my scientific work. I am very grateful for uncountable hours of scientific discussion, for the fruitful cooperation in finishing various research papers, for the helpful comments on my thesis draft, and for the dedication and support he has given me at every stage of my studies.

I would like to thank Prof. Dr. Gregor Engels and Prof. Dr. Le Gruenwald for reviewing this thesis. Thanks also to Prof. Dr. Friedhelm Meyer auf der Heide and Prof. Dr. Ulrich Rückert for co-supervising my Graduate School studies.

During my research, I had the privileged opportunity to collaborate with Prof. Dr. Le Gruenwald (The University of Oklahoma), Prof. Dr. Panos K. Chrysanthis (University of Pittsburgh), and Prof. Dr. George Samaras (University of Cyprus). I benefited very much from their excellent expertise, our productive discussions at Schloss Dagstuhl, and our cooperation on several joint publications.

Furthermore, I would like to thank the International Graduate School of Dynamic Intelligent Systems at the University of Paderborn for establishing a stimulating environment for research and for providing financial support in the form of a scholarship.

To my beloved wife, Uta, and to our beloved daughter Isabel Marie: Thank you for all the love, support, and happiness that you brought to me.

Sebastian Obermeier
Paderborn, March 2008

Contents

Chapter 1 Introduction 1

1.1 Mobile Databases . . . 1
1.1.1 Examples of Mobile Database Applications . . . 2
1.2 Differences between Mobile Ad-Hoc Network and Fixed-Wired Network Transaction Processing . . . 3
1.2.1 Enhanced Failure Model . . . 3
1.2.2 Message Reception Model . . . 4
1.2.3 Device Controllability . . . 4
1.2.4 Compensation Applicability . . . 4
1.3 Problem Description . . . 5
1.4 Roadmap and Bibliographic Notes . . . 5

Chapter 2 Fundamentals 7

2.1 Database . . . 7
2.2 Transaction . . . 7
2.3 Distributed Transaction . . . 7
2.4 Local Transaction Model . . . 8
2.5 Atomicity . . . 8
2.6 Atomic Commit Protocols . . . 9
2.6.1 Two-Phase Commit Protocol . . . 9
2.6.2 2PC Optimizations . . . 10
2.6.3 Three-Phase Commit Protocol . . . 14
2.6.4 Paxos Commit Protocol . . . 17

Chapter 3 Transactions for Web Services 19

3.1 System Model 19
3.2 Web Service Transaction Model 19
3.2.1 Related Transactional Models 22
3.3 Concurrency Control 24
3.3.1 Two-Phase Locking 24
3.3.2 Local Concurrency Control by Backward Validation 25
3.4 Blocking Behavior of Locking and Validation 26
3.5 Summary 27

Chapter 4 Adjourn State 29

4.1 Pre-Atomic Commit Protocol Blocking Problem 29
4.2 Adjourn State Blocking Reduction 30
4.2.1 The Blocking State 30
Blocking State for Locking 30
Blocking State for Validation 30
4.2.2 The Non-Blocking Adjourn State 31
Adjourn State for Locking 31
Adjourn State for Validation 32
4.2.3 Local Restarts and Re-Use of Sub-Transactions 32
4.2.4 Entering the Adjourn State 33
4.3 Number of Messages 35
4.4 Commit Tree 35
4.4.1 An Example of the Coordinator's Commit Tree 36
4.4.2 Commit Tree Modification by the Result Operation 37
4.4.3 Commit Tree Modification by Repetition 39
4.4.4 Benefits of Combining Commit Tree and Adjourn State . . 39
4.5 Experimental Evaluation 40
4.5.1 Scenario 42
4.5.2 Results 44
4.5.3 Evaluation Summary 44
4.6 Related Work 45
4.7 Summary and Conclusion 45

Chapter 5 Cross Layer Commit Protocol 47

5.1 Problem Description . . . 47
5.2 Decentralized Commit Phase . . . 49
5.2.1 Design Goals . . . 49
5.2.2 Key Design Concepts . . . 49
5.2.3 Commit Matrix . . . 50
5.2.4 Merging Commit Matrices . . . 51
5.2.5 Decentralized Commit Phase Algorithm . . . 53
Commit . . . 55
Timeout Attempt . . . 55
Abort Attempt Due to Timeout . . . 56
Abort . . . 56
5.3 Termination Phase . . . 57
5.3.1 Informal Description of the Termination Phase . . . 58
5.3.2 Termination Algorithm . . . 58
5.4 Correctness and Liveness . . . 66
5.4.1 Correctness . . . 66
5.4.2 Liveness . . . 68
5.5 Experimental Evaluation . . . 70
5.5.1 Quasi-Unit-Disc Reception Model . . . 70
5.5.2 Consequences for Atomic Commit Protocols . . . 71
5.5.3 Setup . . . 71
5.5.4 Mobility Models . . . 72
5.5.5 Transaction Generation . . . 72
5.5.6 Routing Protocols . . . 74
5.5.7 Energy Consumption . . . 75
5.5.8 Results . . . 76
5.5.9 Conclusion from the Experiments . . . 82
5.6 Related Work . . . 82
5.7 Summary and Conclusion . . . 84

Chapter 6 Bi-State-Termination 85

6.1 Problem Description . . . 85
6.2 Transaction Model Analysis . . . 86

6.3 Solution . . . 88
6.3.1 Bi-State-Termination . . . 90
6.3.2 Complexity . . . 91
6.3.3 Correctness . . . 92
6.4 BST Rewrite Rules . . . 92
6.4.1 Status Without Active Transactions . . . 93
6.4.2 Write Operations on the BST Model . . . 93
Insertion . . . 93
Deletion . . . 93
Update . . . 94
Set-Oriented Write Operations . . . 94
Completion of a Transaction . . . 94
6.4.3 Read Operations on the BST Model . . . 95
Selection . . . 96
Duplicate Elimination . . . 96
Set Union . . . 96
Projection . . . 97
Cartesian Product . . . 97
Set Difference . . . 97
Other Algebra Operations . . . 98
6.5 Implementation . . . 98
6.5.1 Fast-BST – Write Operations . . . 98
6.5.2 Fast-BST – Read-Operations . . . 100
6.5.3 Commit and Abort . . . 101
6.6 Experimental Evaluation . . . 101
6.6.1 BST Stress Test . . . 101
6.6.2 BST TPC-C Test . . . 104
6.6.3 Evaluation Summary . . . 104
6.7 Related Work . . . 106
6.8 Summary and Conclusion . . . 107

Chapter 7 Summary and Conclusion 109

Bibliography 113

Chapter 1

Introduction

Mobile devices are becoming more capable regarding processing power and connectivity, thus, such devices are used more and more to access and modify data on the go. The advance of ad-hoc network technologies make wireless infrastructures superfluous, which greatly augments flexibility and motivates the ubiquitous use of mobile devices. However, it also gives way to a number of new problems, especially when transaction support is required in order to maintain data consistency and integrity. The handling of unpredictable disconnections of mobile clients and their movement, which may result in network partitioning, requires atomic commit protocols that exhibit short blocking times and are robust against node failures. Furthermore, clients underlie a finite source of energy, which limits the ability to send and receive an arbitrary number of messages.

While traditional transaction models have been designed for a flat transaction invocation, current trends to use service oriented architectures motivate hierarchical transaction invocation models that support additional web service requirements as, for example, dynamic web service orchestration. Thus, mobile nodes can provide and make use of web services, whose mobile database accesses should be encapsulated by spawning database transactions.

Characteristic for our assumed environment is the absence of both a stable database server and a reliable fixed-wired network that the mobile clients can communicate with. Therefore, all communication relies on an unstable radio network using multi-hop wireless routing to maintain communication between at least some of the mobile clients some of the time.

1.1 Mobile Databases

We assume that each device of our mobile ad-hoc network runs a local database that performs the transaction and data management functions. Each client may offer its mobile database content to other participants through a web service. Al-

though a mobile database continues working even when the wireless connection breaks up, it cannot communicate with other nodes, thus it cannot receive service requests or send service results.

1.1.1 Examples of Mobile Database Applications

The following examples, which are also described in [7], give an overview of mobile applications and challenges regarding transaction processing.

M-Commerce scenarios like mobile auction applications assist sellers and buyers, for example of a flea market. Sellers may use their mobile devices to describe their offered goods, while buyers search for and and locate desired items. In addition, the amount of mobile devices allows multi-hop ad-hoc communication. Contracts between several buyers can be signed to gain volume discounts. In this context, transaction support is necessary in order to achieve consistency for contracts and buying/selling actions across all involved databases.

A more detailed description of the design of a mobile flea market application regarding data caching has been published in [11, 54].

Rescue applications use mobile networks to communicate with different machines and human beings. A rescue application for fire fighters can be used to develop rescue plans and form virtual teams of different fire brigades. Furthermore, the mobile networks can be used to locate fire trucks and give moving instructions to them. In such a scenario, there is the need for transaction support in the sense that fire trucks and fire fighters are considered as resources that receive instructions. Since resources cannot perform contradictory instructions at the same time and most plans require that more than one unit processes the instructions, properties like atomicity and isolation must be supported.

Autonomous Rover Vehicles, e.g. Mars rovers, explore new terrains, collect measurements, and take rock samples. In order to analyze the surface of the planet, the Mars rovers must further combine their locally measured data with data that is already present in the network. Thus, the network serves as a large database. While some types of Mars rovers move fast, some of them move at a moderate speed. As the application of well tried and tested standard distributed database technology in such a scenario is desirable, transaction processing must be modified in order to stabilizes the coordination

process, and to reduce the blocking of participating databases, especially if the databases are suspected to frequently disconnect from the network.

Homecare Applications assist nurses with mobile devices like PDAs to get information about patients. Transaction support is required whenever patient data, medical data, or subscriptions get updates. However, the amount of transactions is usually very low. Therefore, the concurrency control mechanisms used do not need to be efficient regarding the transaction throughput, but they must be efficient regarding data blocking.

1.2 Differences between Mobile Ad-Hoc Network and Fixed-Wired Network Transaction Processing

If we compare traditional transaction processing in fixed-wired environments with transaction processing in mobile environments, we can identify the following new challenges that are induced by the mobile character of a network and its underlying applications.

1.2.1 Enhanced Failure Model

Compared to fixed-wired networks, mobile environments suffer from a variety of failures:

Message loss occurs in fixed-wired networks due to rare problems like buffer overflows or data packet collisions. In mobile networks, however, message loss occurs more frequently. For example, if the sender or receiver moves out of scope, if the channel suffers from interference, if obstacles hamper the transmission, or if the sender's or receiver's battery drains suddenly during message transmission.

Network partitioning due to the movement of participants occurs in mobile environments more frequently than in fixed-wired networks, in which this event is very seldom.

Device failure due to low energy may occur frequently in mobile networks.

1.2.2 Message Reception Model

In mobile ad-hoc networks, a message that is sent is not only received by the destined recipient. Each participant that is close to the sender can hear the message. When a routing strategy is used, participants that are located close to the path that the message takes will get the message. We will see in Chapter 5 how our Cross Layer Commit Protocol utilizes this characteristic.

1.2.3 Device Controllability

Distributed databases in fixed-wired networks are usually used for performance and availability reasons. Thus, there is often a single database owner that controls all of its databases. In mobile networks, however, each user owns and controls only a single device. Thus, whenever distributed transactions must access multiple devices, we cannot guarantee that all individual users cooperate and do not move away, since there is no central instance that controls the devices.

1.2.4 Compensation Applicability

Transaction processing models that apply the concept of *compensation*, e.g. [23, 56, 57], explicitly allow databases to run into a possibly inconsistent state that is compensated later. However, the models using compensation assume that databases are somehow connected to a single site that controls the compensation. Thus, participants having inconsistent states will not participate in proceeding transactions with different participants as long as their inconsistent states have not been compensated.

However, if a network contains independent mobile participants that are susceptible to network partitioning, there is no guarantee that a compensating transaction will be received by the destined database. This means, that a database may contain inconsistent data, but has no knowledge of this inconsistency. Furthermore, the database may participate in transactions that are based upon this inconsistent data with different participants. As a result, the state of inconsistency is passed on to other participants. Since this "chain of inconsistency" may be arbitrarily long and compensation is not possible if participants cannot be reached, e.g. if the network is partitioned, protocols relying on compensation cannot guarantee atomicity in mobile ad-hoc environments where participants are autonomous.

1.3 Problem Description

In this thesis, we focus on the following major problems:

1. During the execution of a web service on multiple participant nodes, the nodes wait for the atomic commit protocol to start and *block* the used resources for a predefined time. When the time limit has exceeded before a vote message has arrived at the client, the transaction is aborted. The problem is that setting up a timeout that enhances the number of committed transactions and reduces the overall blocking time is extremely difficult since it depends on the network quality that is not necessarily constant over time.

2. During the atomic commit protocol, a sequence of failures may lead to a situation where the atomic commit protocol instance cannot terminate with a unique commit or abort decision. As traditional atomic commit protocols consist of only a single commit coordinator and rely on a stable network, atomic commit protocol blocking occurs more frequently when the coordinator fails or the network is partitioned. The problem of other contributions using multiple commit coordinators is the higher energy consumption that occurs due to a massive increase of messages.

3. After a database proposed to execute a transaction T by sending a voteCommit message, the database may not receive the final commit decision. In this situation, *transaction blocking*, which summarizes the unilateral impossibility to abort or commit a transaction, occurs. Transaction blocking can lead to situations in which resources are not useable by other transactions for an infinitely long time.

1.4 Roadmap and Bibliographic Notes

We start by explaining fundamental transaction concepts in Chapter 2, including a new web service oriented transaction model (Chapter 3). Parts of this transaction model, which has been especially designed for mobile environments, are published in [8, 10, 13].

During the execution of the web service, the participating nodes wait for the atomic commit protocol to start. During this wait, the *Adjourn state* allows to reduce the blocking and increases the transaction throughput. In Chapter 4, we describe this Adjourn state, which has been published in [51,52]. The Adjourn state is based on the *Suspend state* that has been published in [8, 13].

During the execution of the atomic commit protocol within a mobile ad-hoc environment, several failures can occur that lead to atomic commit protocol blocking. In Chapter 5, we describe the *Cross Layer Commit Protocol*, a failure tolerant atomic commit protocol that makes use of mobile network characteristics and reduces the required energy consumption. This atomic commit protocol is described in [53]. It is based on the *Multiple Coordinator Protocol (MCP)* of [9, 12].

Even though our atomic commit protocol tolerates network problems, some nodes may not receive the transaction decision when the nodes have moved. The concept of *Bi-State-Termination*, first published in [50], allows participants to unblock resources, which means that participants can even execute conflicting transactions. We explain this concept in Chapter 6.

Finally, Chapter 7 summarizes and concludes this thesis.

Chapter 2

Fundamentals

This chapter gives definitions for fundamental transaction processing terms.

2.1 Database

A *database* consists of a set of *data tuples*. Each of these data tuples has a *value*. A *database system* describes a collection of modules that access the database. We call each module that performs such an access a database *operation*. The simplest operations are *read* and *write* operations. The read-operation returns the value of a data tuple, while the write operation allows to change the data tuple's value.

2.2 Transaction

A *database transaction* is a sequence of (read- and write-) operations that is treated as a single operation. Thus, a database transaction must be executed in an atomic fashion, which means that all of its operations must be either entirely *committed*, or all of them must be *aborted*. The effects of a committed transaction become permanent, while an aborted transaction must not have any effect on the database.

There are several reasons why a database must abort a transaction, e.g. if the transaction violates consistency constraints, if the transaction conflicts with concurrent transactions, if required locks are not available for a certain time, or if the database is running out of main memory or hard disk space.

2.3 Distributed Transaction

A *distributed transaction* is a transaction in which two or more network hosts are involved. The set of operations processed at a single host is called a *sub-transaction*.

Compared to non-distributed transaction processing, the processing of distributed transactions requires consideration of an enhanced failure model. For example, message loss and node failure must be handled when processing distributed transactions.

2.4 Local Transaction Model

During the execution of a sub-transaction, a database enters the following phases: the *read-phase*, the *commit decision phase*, and, in case of successful commit, the *write-phase*. While executing the read-phase, each sub-transaction invokes necessary sub-transactions and carries out write operations on its private transaction storage only. During the commit decision phase, the participating databases use an *atomic commit protocol* to decide on the transaction's commit decision, which can be either *commit* or *abort*. If the transaction's commit decision is abort, each database discards all changes made by the corresponding sub-transaction. If the transaction's outcome commit decision is commit, the database executes the sub-transaction's write phase. During this phase, the private transaction storage is transferred to the durable database storage, such that the changes done throughout the read-phase become visible to other transactions after completion of the write-phase.

2.5 Atomicity

Atomicity in the field of transaction processing means that a transaction T_i including all of its operations, either runs successfully to completion, or, if T_i does not complete, that T_i has no effect at all and leaves the database in a state as if the transaction had never been started.

Guaranteeing atomicity for non-distributed transactions is a task that is fulfilled by the database itself.

However, if atomicity must be guaranteed for a distributed transaction T_i, it is required that all sub-transactions belonging to T_i commit or all sub-transactions belonging to T_i abort. Thus, a single sub-transaction cannot immediately commit at a host after is has been executed, since other sub-transactions belonging to the same transaction might abort and an abort of a single sub-transaction requires all other sub-transactions to abort as well. Thus, to achieve atomicity for distributed transactions, communication between the participants is necessary. We call the

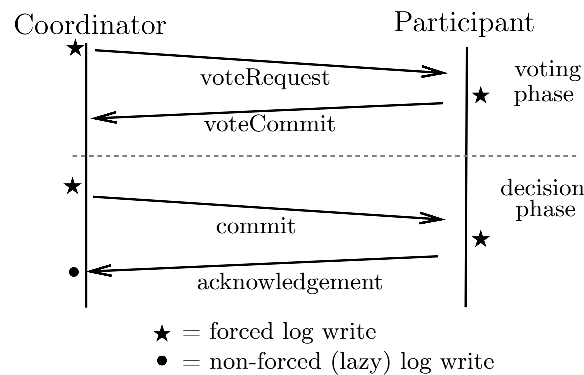

Figure 2.1: 2-Phase Commit Protocol

protocol that leads all participating databases to an atomic decision an *atomic commit protocol*.

2.6 Atomic Commit Protocols

We discuss two traditional atomic commit protocols, Two-Phase Commit (2PC) and Three-Phase Commit (3PC), which are mainly used within fixed-wired environments. Furthermore, we describe the recently proposed Paxos Commit Protocol that derives a commit decision based on the Paxos Consensus algorithm.

2.6.1 Two-Phase Commit Protocol

The basic Two-Phase Commit Protocol (2PC), first described by [28] and [46], is the first protocol that was proposed to guarantee atomicity for distributed transactions. As the name implies, 2PC consists of two phases. During the first phase, called *voting phase*, the coordinator demands a vote on the transaction's commit decision in terms of "voteAbort" or "voteCommit" from each participant. When all participants have voted, the coordinator proceeds to the second phase, called *decision phase*, in which the coordinator decides for "commit" of the transaction if all votes are "voteCommit", otherwise the coordinator decides for "abort". Figure 2.1 illustrates the protocol.

A participant that voted for "commit" is not allowed to abort or commit the transaction on its own until it has received the coordinator's commit decision. A participant that voted for "abort" can immediately abort the transaction, which means, the database must restore a state that is equal to a situation in which

the transaction has never been executed, i.e. all changes caused by the aborted transaction must be rolled back.

If a database has voted for "commit" and the decision on the transaction is "commit" as well, the transaction's changes become durable, i.e. the write phase is executed, and thus the transaction's changes are visible to other transactions. Furthermore, when the write phase is completed, locks held by the completed transaction can be released. After the transaction has successfully committed, each participant sends an acknowledgement to the coordinator. If the decision on the transaction has been "abort", the database aborts the transaction as described above.

2PC involves several logging activities in order to deal with communication and system failures. The coordinator force-writes an initialization record before it requests the votes, which is a log entry that is forced to be written immediately to the stable storage. Then, each participant force-writes a "voteCommit" entry before it sends the "voteCommit" message to the coordinator, or, when the transaction must be aborted, the participant force-writes an abort record. Before the coordinator sends the final commit decision, it force-writes a log entry about the "commit" or "abort" decision. When the decision is received by a participant, an additional log entry that the transaction is going to be committed or aborted respectively is required before an acknowledgement is sent to the coordinator. The coordinator forgets about the transaction after all acknowledgements have been received, since the coordinator is sure that each participant has written the commit or abort decision into its log, and thus no participant will any more inquire about the transaction's commit decision. Therefore, the coordinator issues a non-forced (lazy) log write that logically eliminates the transaction from the log. For n participants, the basic 2PC requires $2n + 2$ forced log writes and $4n$ messages.

In the next section, we discuss two further logging strategies for 2PC, namely the Presumed Abort Protocol and the Presumed Commit Protocol optimization that are used depending on the volume of the transactional system and the assumed frequency of failures.

2.6.2 2PC Optimizations

As atomic commit protocols are traditionally used for guaranteeing the integrity of distributed databases within environments where physically different databases are connected by high-speed networks, overall database performance is an important criterion for high-volume transactional systems. As [64] has evaluated, the transaction performance decreases when using 2PC due to the increased costs of

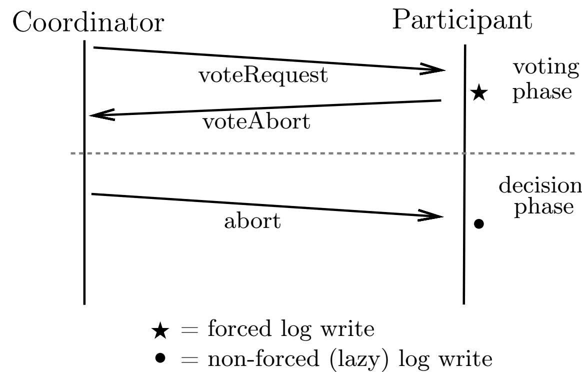

Figure 2.2: Presumed Abort 2PC, Abort Case

forced log writes. Thus, one goal of 2PC optimizations is to reduce the number of forced log writes by designing more clever algorithms that restore the state of the atomic commit protocol in case a participant suddenly reboots or looses main memory information.

The *Presumed Commit Protocol* and the *Presumed Abort Protocol* have been proposed in [47,48]. Both protocols are optimizations of 2PC and presume that the transaction will commit (abort), thus reducing the number of forced log writes.

The idea of Presumed Abort is illustrated in Figure 2.2, which shows the abort case. In case the coordinator decides for abort, it does not write any log record for the following reason: Whenever the coordinator suddenly looses its main memory information and must reboot, it does not know anything about the transaction. If a participant inquires about the transaction's commit decision, the coordinator will not find a log entry; thus, the coordinator answers with "abort". Therefore, the first log entry of a coordinator can also be skipped, since in case of failure, the decision for the transaction would be abort.

The commit case, illustrated in Figure 2.3, requires the coordinator to force-write a commit record before the commit command is sent to the first participant in order to ensure that the coordinator will not mistakenly presume abort after the coordinator has failed and restarted.

The commit case of *Presumed Commit* is illustrated in Figure 2.4. In contrast to Presumed Abort, the idea of Presumed Commit is that a coordinator decides for commit whenever no information about a transaction can be found within the log. In contrast to Presumed Abort, this requires the coordinator to force-write an initialization record before the votes are requested to indicate that the transaction

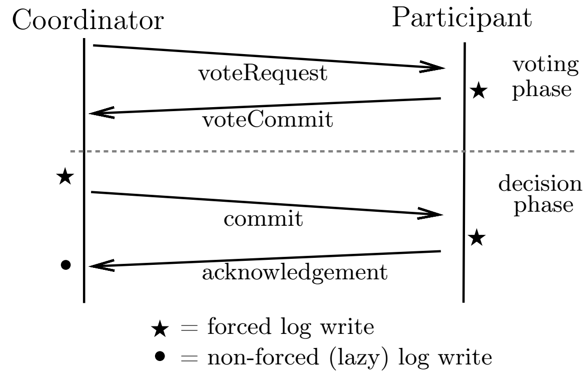

Figure 2.3: Presumed Abort 2PC, Commit Case

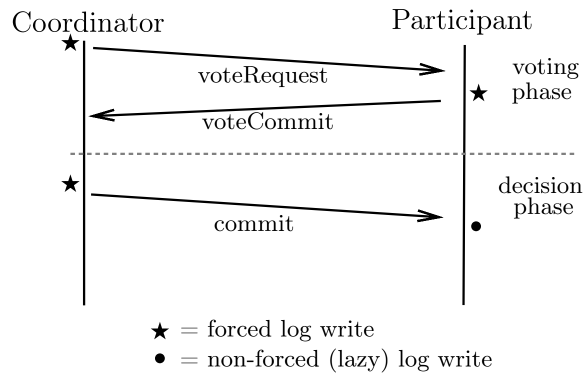

Figure 2.4: Presumed Commit 2PC, Commit Case

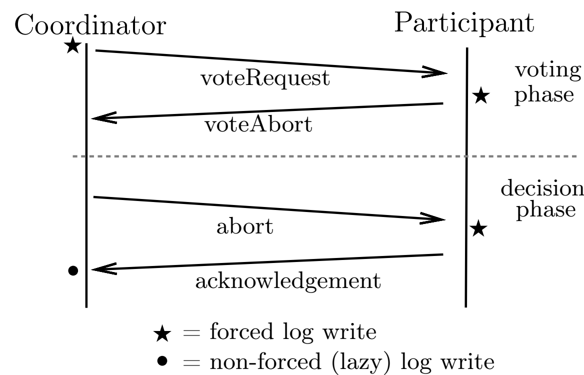

Figure 2.5: Presumed Commit 2PC, Abort Case

coordination is still in progress and has not been committed. However, when the decision for commit can be made, the coordinator must forget about the transaction before it sends the commit command, since an inquiring participant will always receive the answer "commit" when no log entry on the transaction is found. Thus, the coordinator force-writes an entry that logically eliminates the initialization record before sending the commit command. The benefit of Presumed Commit is that in case of commit, an acknowledgement message from each participant is not necessary anymore.

The abort case of Presumed Commit is illustrated in Figure 2.5. In this case, an acknowledgement message of each participant is necessary. After each participant has acknowledged, the coordinator can be sure that each participant has written the decision into its log and thus, no participant will inquire about the transaction in the future anymore.

In comparison with standard 2PC, the Presumed Abort optimization saves in total n messages (the acknowledgement messages) and $n + 2$ forced log writes for an aborted transaction that involves n participants, but saves only one forced log write for committed transactions. In contrast, Presumed Commit saves n messages and n forced log writes in case of commit (the participant's commit entries), but saves only one forced write in case of abort. A detailed description of both protocols can also be found in [37, 71].

Based on the Presumed Commit Protocol, [42] proposed the *new Presumed Commit* Protocol, which eliminates the forced write of the initialization record in the beginning of transaction coordination. Since this omission leaves the coordinator unclear about all active transactions in case of a failure, an additional set of *poten-*

tially initiated transactions is required, which is calculated by means of increasing transaction IDs. Whenever the transaction having the lowest transaction ID has been committed or aborted, a lower boundary of open transactions is calculated and written to the log. However, in contrast to Presumed Commit, crash related information must be kept even in case of commit.

[65] proposes the *unsolicited-vote* optimization, which is based on the assumption that each participant knows when it has finished all operations belonging to a transaction. Thus, a participant can immediately send a vote in conjunction with the transaction's result. Although this optimization makes the "voteRequest" message superfluous, each participant looses the right to abort a transaction due to a timeout, since a participant is doomed to wait until it has received the coordinator's commit decision. In contrast to standard 2PC, the unsolicited-vote optimization is more susceptible to participant and coordinator failures, since the blocking time of participants increases with varying transaction execution times.

2.6.3 Three-Phase Commit Protocol

A major problem of 2PC and its optimizations is that 2PC involves a *waiting state*, which participants enter after they have sent the vote on the transaction to the transaction initiator and are waiting to receive the decision from the coordinator. For 2PC, the set of possible successor states that follow the waiting state contains both, the abort state and the commit state. A coordinator failure including a participant failure can lead to a blocking situation in which the remaining participants cannot decide for abort or commit, since the failed participant may have advanced to either the abort or commit state. Even a new coordinator cannot terminate the transaction when at least one participant's state is unknown.

The Three-Phase Commit Protocol (3PC) was designed by [62] to overcome this problem in environments in which site failures can occur but not network partitioning. The main idea of 3PC is to use an additional *dissemination phase* before a transaction is finally committed. Figure 2.6 illustrates a successful commit coordination for 3PC. The state chart for 3PC showing the coordinator can be found in Figure 2.7, while the state transitions for a participant are illustrated in Figure 2.8.

In the first phase of the 3PC, the coordinator demands the votes of the participants and proceeds to the state w_i. A participant that has received the message "voteRequest" answers with either commit or abort, depending on the transaction's read-phase outcome, and also proceeds to the state w_i. If, and only if all vote messages were commit, the coordinator sends a "prepareToCommit" message to each

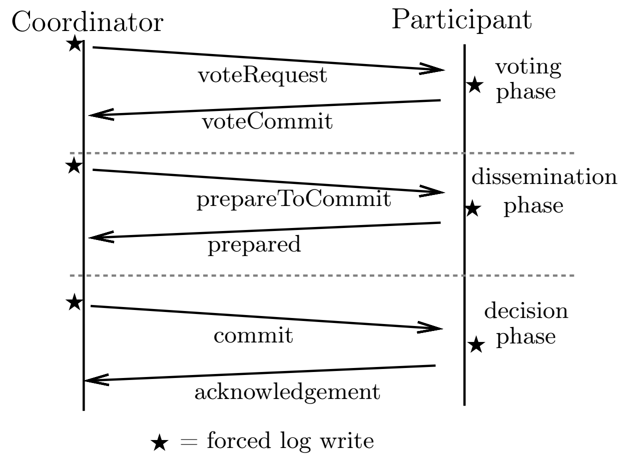

Figure 2.6: 3-Phase Commit Protocol

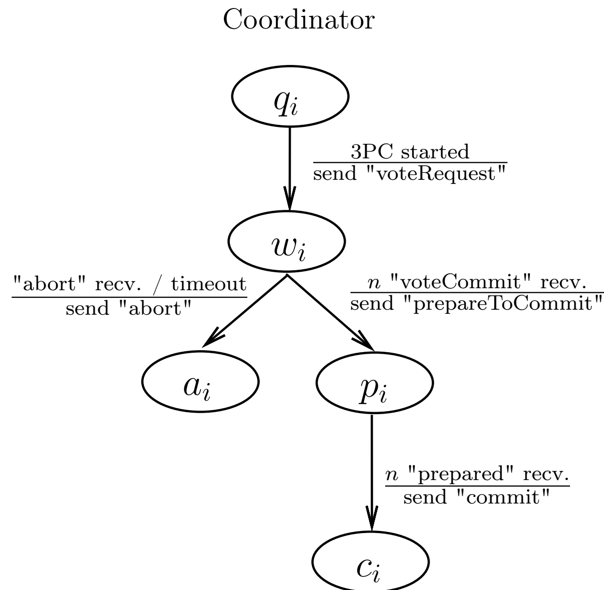

Figure 2.7: 3PC States (Coordinator, n Participants)

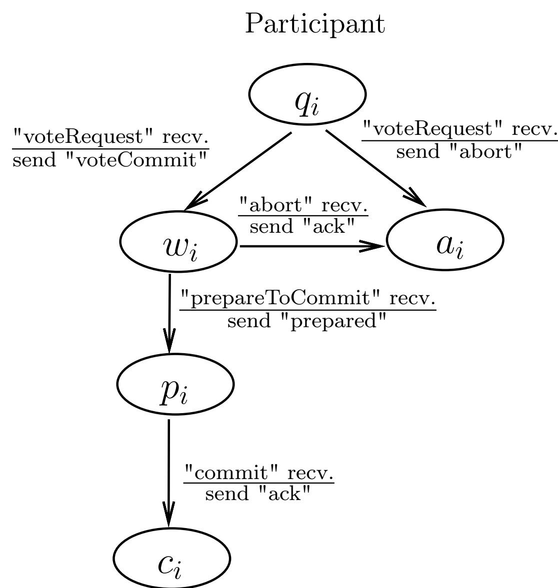

Figure 2.8: 3PC States (Participant)

participant and proceeds to the state p_i. A participant that has received the "prepareToCommit" message acknowledges by sending "prepared" and proceeds to the state p_i. Again, the coordinator needs the "prepared" messages of all participants to instruct them to commit the transaction and proceed to the state c_i.

Note that a participant that has sent a "voteCommit" message is no longer allowed to abort the transaction on its own. Therefore, the message pair ("prepareToCommit", "prepared") is previously fixed. Although, at first, these messages seem to be superfluous, 3PC guarantees that for any two participants, which may differ in at most one state, the set of possible successor states does not contain an abort and commit simultaneously. In case of coordinator failure, this allows the participants to elect a new coordinator that *terminates* the transaction. This newly elected coordinator first queries all available participants about their last state within the protocol execution. Whenever at least one participant has received a "prepareToCommit" message, i.e. it is in the state p_i, the new coordinator decides for commit. In this situation, all participants have voted for commit, and no participant may have aborted the transaction since a participant can be at most in the waiting state w_i. If, in contrast, all participants are within the waiting state and none in the prepared state p_i, no participant can have committed the transaction, i.e. is in the

state c_i, but some participants may be in the state a_i. Thus, the new coordinator will decide for abort.

The termination phase for 3PC is only correct if no network partitioning has occurred. Otherwise, two new coordinators of two different network partitions may come to different decisions on the transaction. Thus, whenever mobile networks are susceptible to network partitioning, the termination mechanism of 3PC can lead to wrong results.

2.6.4 Paxos Commit Protocol

The Paxos Commit Protocol [30] is based on the Paxos Consensus approach described in [40,41], which was originally designed to let participants agree on a single decision among several options. The main idea of this approach is that whenever a majority of participants have accepted a decision D, this decision becomes implicitly anchored among all participants. This anchorage guarantees that any further proposal will be the same as the anchored decision D.

Paxos Commit uses multiple coordinators by assigning them different roles. One of them has a special role, it is called the *leader*, the other coordinators are called *acceptors*. Each participating database forwards its commit vote to each of the Paxos acceptors. Each acceptor then forwards its collected database votes to the leader, which determines whether the transaction must be committed or aborted, or, if some acceptors have not received some of the database votes, whether some acceptors must be additionally notified. When the leader's proposal has been made, it is forwarded to the acceptors and to the databases.

To ensure that the protocol progresses even if the leader fails, each acceptor may decide for itself if and when it becomes a leader. To handle conflicts when more than one leader is present, Paxos uses increasing *version numbers* to identify the highest leader and the corresponding leader's proposal. An acceptor accepts a new proposal only if the new proposal has a higher version number than the old proposal. A new leader must build a proposal by adopting the previous proposal having the highest version number.

In contrast to the termination phase of 3PC, Paxos Consensus works correctly even when network partitioning occurs.

The termination phase of our CLCP protocol, which is based on Paxos Commit, gives a more detailed explanation of the use of version numbers and majorities in Chapter 5.

Chapter 3

Transactions for Web Services

3.1 System Model

We consider mobile devices, each equipped with a local database. The mobile devices form a mobile ad-hoc network and offer their data by providing (web-) services. When a device uses an offered Web service of a participant P_i, one or more sub-transactions are invoked at the local database of P_i, which may invoke other sub-transactions on mobile devices P_k.

Due to the distributed character of our system model, one challenge when guaranteeing the atomicity property is to block resources as shortly as possible.

3.2 Web Service Transaction Model

In contrast to traditional flat transaction models [6], in which a transaction is split a-priori into several sub-transaction's, our service oriented approach is based on the "Web Services Transactions Specification" [14]. However, due to our focus on the blocking problem when guaranteeing atomicity, we can use a much simpler transaction model in this paper, i.e., we do not need a certain Web service modeling or composition language like BPEL4WS [20]. Thus, our transaction model consists only of the objects "application", "transaction procedure", "Web service", and "sub-transaction". These terms are related to each other as explained in the following.

An *application* AP may consist of one or more *transaction procedures*. A transaction procedure is a Web service that must be executed in an atomic fashion. Transaction procedures and Web services are implemented using local code, database instructions, and (zero or more) calls to other remote Web services. Since the invocation of a Web service depends on conditions and parameters, different executions of the same Web service may call different Web services and execute different local code.

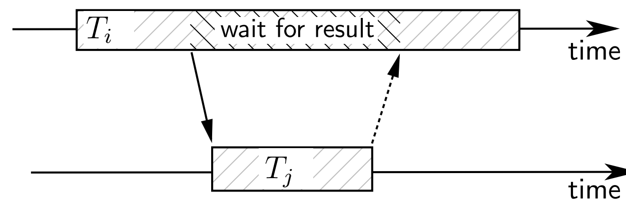

Figure 3.1: Synchronous Calls for Web Services

When AP executes a transaction procedure, we call AP the *Initiator*, and we call the execution of the transaction procedure a *transaction* T_i. The application AP is interested in the result of T_i, i.e. whether the execution of the transaction T_i has been committed or aborted. In case of commit, AP is also interested in the return values of the parameters of T_i.

The relationship between transactions, Web services, and sub-transactions is recursively defined as follows: We allow each transaction or sub-transaction to dynamically invoke additional Web services offered by physically different nodes. We call the execution of such Web services invoked by a transaction or sub-transaction T_i the sub-transactions $T_j \ldots T_n$ of T_i, respectively. This invocation hierarchy can be arbitrarily deep. Thus, the sub-transaction T_j can invoke another sub-transaction T_k, as well. However, if we do not need to refer to the relationship between T_j and T_k, we call both of them transactions.

Whenever $T_j \ldots T_n$ denote all the sub-transactions called by either T_i directly or by any child or descendant sub-transaction T_k of T_i during the execution of the transaction T_i, atomicity of T_i requires that either all transactions of the set $\{T_i, T_j, \ldots, T_n\}$ commit, or all of these transactions abort.

We assume that each Web service only knows the Web services that it calls directly, but does not know whether or not the called Web services call other Web services. Therefore, at the end of its execution, each transaction T_i knows which sub-transactions $T_j \ldots T_n$ it has called, but T_i, in general, will not know which sub-transactions have been called by $T_j \ldots T_n$. Furthermore, we assume that usually a transaction T_i does not know how long its sub-transactions $T_j \ldots T_n$ are going to run.

In the mobile architecture for which our protocol is designed, Web services are invoked by asynchronous messages instead of invoking them by synchronous calls for the following reason. We want to avoid dependencies that occur if the completion of the read-phase of a (sub-) transaction T_i depends on the execution of another

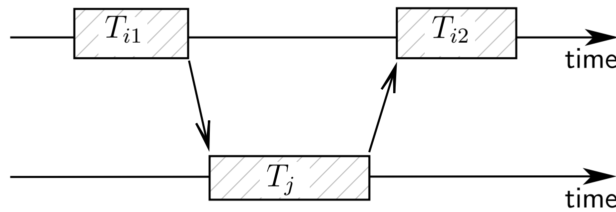

Figure 3.2: Modelling Synchronous Web Service Calls by Asynchronous Invocations

sub-transaction T_j. Figure 3.1 illustrates this situation: The Web service T_i must wait for the result of T_j before it can finish its execution.

In contrast, we want each sub-transaction to be able to autonomously complete its read phase. Thus, we allow sub-transactions only to return values indirectly by asynchronously invoking corresponding receiving Web services, and not synchronously by return statements. However, our model also supports a synchronous Web service call of T_i to T_j by modeling the call as Figure 3.2 illustrates: we split T_i into T_{i1} and T_{i2} as follows. T_{i1} includes T_i's code up to and including an asynchronous invocation of its sub-transaction T_j; and T_{i2} contains the remaining code of T_i. T_j additionally contains an asynchronous call to T_{i2} that contains the return values computed by T_j that shall be further processed by T_{i2}.

Since (sub-)transactions describe general services, the nodes that execute these (sub-) transactions may be arbitrary nodes and are not necessarily databases. Thus, we also call these nodes *resource managers (RM)*.

One feature of our Web service transactional model is that the Initiator and the Web services do not need to know in advance every sub-transaction that is generated during transaction processing.

Figure 3.3 shows an example execution of our Web service transaction model. The Initiator of the transaction invokes a Web service T_i that is offered by participant P_i. During the completion of the read-phase, an additional transaction T_j that is offered by participant P_j is necessary to fulfill T_i. Furthermore, the Initiator is notified whenever a sub-transaction's read-phase is completed, as the dotted arrows indicate. We will see the benefit of this notification regarding a fast determination of all involved participants in Section 4.4. Figure 3.3 additionally shows how call-by-value invocations are implemented: When Participant P_i invokes the sub-transaction T_j at P_j, it additionally adds the parameter $[T_k]$ to specify that T_k must be invoked with the corresponding return value when T_j has finished. Thus, P_j invokes T_k with the requested return value after P_j has finished T_j's read-phase. The Initiator starts

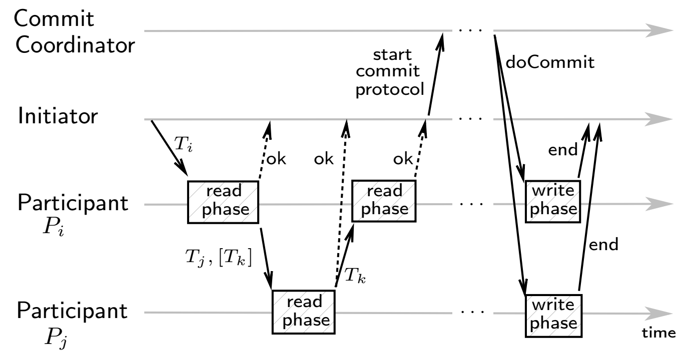

Figure 3.3: Web Service Transaction Execution Sequence

the commit protocol when all sub-transactions have finished, which is explained in Section 4.4. The commit protocol then needs several message exchanges in order to decide on the transaction's commit status.

3.2.1 Related Transactional Models

Our model differs from other models that use nested transactions (e.g. [14,23,45, 55–57]) in some aspects including but not limited to the following:

Since network partitioning makes it difficult or even impossible to compensate all sub-transactions, we consider each sub-transaction running on an individual resource manager to be non-compensatable for the following reason. Committed transactions can trigger other operations, thus, we cannot assume that compensation for committed transactions in mobile networks is always possible, since network partitioning makes nodes unreachable but still operational. When the compensation transaction does not reach the node, however, a model relying on compensation cannot give hard global atomicity guarantees as defined in [35]. Thus, we focus on a transaction model, within which atomicity is guaranteed for distributed, non-compensatable transactions. Therefore, no sub-transaction is allowed to commit independently of the others or before the commit Coordinator guarantees that all sub-transactions can be committed.

[23] is another contribution that relies on compensation and sets up a transaction model called "The Kangaroo Model". Common with this model, we have a global transaction as well as sub-transactions that are created during transaction execution and cannot be foreseen.

Models like [56] or [57] allow a transaction to define the level of consistency which the transactions leave behind. In contrast, our solution does not need to

adjust the level of consistency. We can guarantee atomicity without leaving states of inconsistency, even within mobile environments without base stations, where nodes that have consistent data may crash or permanently remain in a separated network partition.

Different from CORBA OTS [45, 55], we assume that we cannot identify a hierarchy of commit decisions, where aborted sub-transactions can be compensated by executing other sub-transactions. Although we assume that Web services invoke other Web services and the coordinator uses a tree structure to maintain information about commit votes, we do not propose hierarchical commit decisions, since this implies that the upper nodes of the execution hierarchy must wait for the commit decision of all descendant nodes. In a mobile environment, where node failures are likely, our solution allows to spread the commit decision as fast as possible by flattening the invocation tree even before the atomic commit protocol starts.

Web services and their description languages (e.g. BPEL4WS [20] or XLANG [67]) are used more and more to implement nested Web service transactions, which are called *Web services orchestration*. However, these languages do not provide a coordination framework to implement atomic commit protocols.

Different from the Web service transaction model [14], the Initiator of a transaction in our model does not need to know all the transaction's sub-transactions in advance. We assume that each sub-transaction notifies the Initiator after completion of the read-phase by including a list of asynchronously invoked sub-transactions into its "read phase completed" notification message to the Initiator. Thus, our Web services may consist of control structures, e.g. `if <Condition> then <T1> else <T2>`. This means that an execution of a sub-transaction may create other sub-transactions dynamically. These dynamically created sub-transactions belong to the global transaction and must be guided to the atomic commit decision as well. Furthermore, we assume message-oriented communication, i.e., a Web service does not explicitly return a result, but may invoke a receiving Web service that performs further operations based on the Web service result.

The approaches [21] and [22] suggest the suspend state, which unblocks resources as well. However, these approaches are intended for the use within an environment with a fixed network and several mobile cells, where disconnections are detectable and therefore transactions can be compensated. In contrast, our assumed environment, which allows ad-hoc communication, demands a more complex failure model that takes network partitioning into consideration. This means, our model assumes that a coordinator cannot distinguish whether another node has failed or is

still operational in another partition, and therefore compensation cannot be used. In addition, our transactional model is more powerful since it allows dynamically invoking Web service transactions, which need not to be known in advance.

3.3 Concurrency Control

We describe two common concurrency control methods, namely *Two-Phase Locking (2PL)* and *Validation*, which can be used in our Web service transaction model.

For each transaction T_i, let $RS(T_i)$ denote the local data read by T_i, and let $WS(T_i)$ denote the local data written by T_i.

Definition 3.3.1 Two operations O_i and O_j conflict $\iff \exists$ tuple $t \exists$ attribute a : $(O_i$ accesses $t.a \wedge O_j$ accesses $t.a \wedge ((O_i$ writes $t.a) \vee (O_j$ writes $t.a)))$

Definition 3.3.2 A transaction T_j *depends* on T_i if and only if on a database D at least one operation O_i of T_i conflicts with an operation O_j of T_j, and O_i precedes O_j.

Definition 3.3.3 The *serialization graph* of a set of transactions contains the transactions as nodes and a directed edge $T_i \rightarrow T_j$ for each pair (T_i, T_j) of transactions for which T_j depends on T_i.

Definition 3.3.4 *Serializability* requires the serialization graph of all committed transactions to be acyclic.

3.3.1 Two-Phase Locking

The *Two Phase Locking Protocol* [24] consists of two consecutive phases for handling transaction locks:

- In Phase 1, necessary locks are acquired, no lock is released.
- In Phase 2, no lock can be acquired anymore, but locks may be released.

For transactions that obey 2PL, the serializability property is guaranteed according to [68]. However, in order to avoid cascading aborts and to guarantee recoverable histories, we require transactions to be *strict* [6], i.e. Phase 2 can only be entered when the resource manager has received the transaction's commit or abort command. In other words: Unless the transaction's commit decision is not known by a resource manager, the resource manager is not allowed to use the transaction's resources for other purposes. We assume for the remainder of the thesis that the strictness property holds when 2PL is used.

3.3.2 Local Concurrency Control by Backward Validation

Optimistic concurrency control [32, 39], more precisely *backward oriented optimistic concurrency control* with parallel validation, does not use locking and allows parallel access to conflicting data tuples. However, after the read-phase has been finished, an additional *validation phase* follows in which concurrency conflicts are discovered, and only those transactions that are validated correctly may, after a successful distributed commit decision, enter a write phase, within which they write their changes back to the database.

A local sub-transaction T_o is called *older* than a local sub-transaction T_v running on the same database, if T_o starts its *validation phase* before T_v does.

A transaction T_v validates to *true*, if one of the following conditions holds for each older transaction T_o:

1. T_o has completed its write phase before T_v has started.
2. T_o has completed its write phase after T_v has started but before T_v has started its validation phase, and $(RS(T_v) \cap WS(T_o)) = \varnothing$.
3. T_o has not finished its write phase before T_v has started the validation, and
   (a) $(RS(T_v) \cup WS(T_v)) \cap WS(T_o) = \varnothing$, and
   (b) $(RS(T_o) \cap WS(T_v)) = \varnothing$

Compared to [39], we additionally require condition 3(b) to be fulfilled in order to guarantee serializability for distributed transactions. While the concurrency control proposed by [39] correctly guarantees serializability for non-distributed transactions, it cannot guarantee serializability when distributed transactions are executed concurrently.

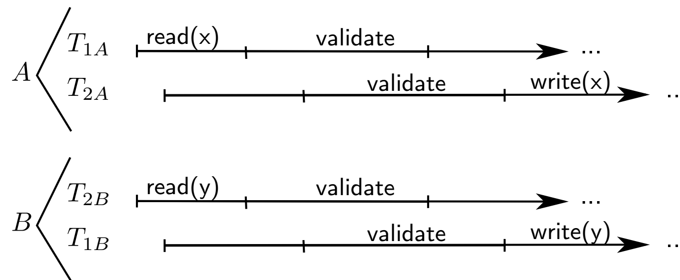

Figure 3.4: Serializability for Distributed Transactions

Figure 3.4 shows a schedule for the participants A and B, each of them running two sub-transactions concurrently. On participant A, a dependency T_{1A} before

T_{2A} exists, while on participant B, a dependency T_{2B} before T_{1B} exists. If a global transaction T_1 consists of the sub-transactions T_{1A} and T_{1B} and a global transaction T_2 consists of the sub-transactions T_{2A} and T_{2B}, the serialization graph would contain a cycle, which violates serializability. The request for checking condition 3(b) prevents this cycle, since our proposed concurrency control protocol would abort both sub-transactions T_{2A} and T_{1B}.

3.4 Blocking Behavior of Locking and Validation

Traditional validation [39] is usually considered a scheduling technique for synchronization of concurrent transactions that avoids blocking.

However, although validation does not directly block any resources, we argue that even the validation-based concurrency control shows a blocking behavior when used in combination with an atomic commit protocol. More precisely, in case of link failures or node failures, locking and validation are equivalent regarding their blocking behavior in the following sense. Assume that a sub-transaction T_i, reading the tuples $RS(T_i)$ and writing the tuples $WS(T_i)$, is waiting for the Coordinator to demand a vote on the transaction.

Two-phase locking would not allow any sub-transaction T_k with $WS(T_i) \cap (WS(T_k) \cup RS(T_k)) \neq \emptyset$ to get the required locks and would therefore block T_k and prevent the completion of T_k's read-phase.

Traditional validation (e.g. [39]) *would allow* any younger sub-transaction T_k with $WS(T_i) \cap (WS(T_k) \cup RS(T_k)) \neq \emptyset$ to enter the read-phase. However, since the tuple sets $WS(T_i)$ and $(WS(T_k) \cup RS(T_k))$ are not disjoint, the validation of T_k fails, resulting in an abort of T_k. Thus, validation prevents T_k to enter its write phase as well. Note that even a repetition of T_k would result in an abort as long as T_i waits for the Coordinator's decision.

This means that both techniques, locking and validation, show a similar behavior when dealing with atomic commit decisions for mobile networks: A transaction T_i that waits for the commit decision and that has accessed the tuples $WS(T_i) \cup RS(T_i)$ during its read-phase, is not allowed to unilaterally commit or abort the transaction. Thus, T_i prevents other sub-transactions T_k that write on the data T_i accessed or that read data T_i has written from being committed.

3.5 Summary

In this chapter, we have introduced a new Web service transactional model suitable for mobile networks, which allows a Web service to dynamically invoke other Web services to fulfil its own service. We have discussed two concurrency control mechanisms for this Web service model, locking and validation, and we have demonstrated the blocking effect that both concurrency control schemes involve.

Chapter 4

Adjourn State

In this chapter, we present a technique that reduces the blocking problem before the atomic commit protocol starts. Our technique does not rely on the difficult setup of reasonable timeouts. In addition, we propose a technique that discovers all dynamically invoked sub-transactions of a Web service.

4.1 Pre-Atomic Commit Protocol Blocking Problem

Regardless of whether validation or locking is used, the following problem occurs when the database is still able to abort a sub-transaction T_i, but the commit coordinator is not reachable anymore: The concurrency control prevents conflicting transactions T_c from being successfully executed. In other words, any delay in the commit phase of T_i has a blocking effect on concurrent conflicting transactions T_c. To solve this problem, [62] has introduced time-outs after which the database aborts the transaction T_i if it is still allowed to do so, i.e. if it has not sent its vote message.

However, especially in mobile networks, the question arises: "What is a reasonable time-out after which the database should abort the transaction T_i if it has not sent a vote message yet?". If the time-out is too large, it prevents concurrent and conflicting transactions T_c from a successful validation, since T_c will not pass the validation phase successfully due to the pending transaction T_i. If the time-out is too short, T_c may be unnecessarily aborted, e.g. when the delay is caused by the network or when the duration of the validation phase differs for the databases participating in the global transaction. Determining a reasonable time-out is difficult since it involves not only knowledge about the network conditions, e.g. device movement, message delivery times, message loss rates, etc., it must also consider the device's computing power and CPU utilization, and the varying duration of the validation phase for each mobile device. Therefore, our solution, which does not rely on such a participant time-out, is much easier to setup, and we will even

see that it increases the overall transaction throughput and reduces the amount of blocking.

4.2 Adjourn State Blocking Reduction

Our solution consists of two parts, namely the *Adjourn state* and the *Commit tree*. The Adjourn state avoids setting up participant time-outs for aborting a transaction by distinguishing between two states in which a database can wait for the coordinator's voteRequest message: the *blocking state* (defined in Section 4.2.1) and the non-blocking *Adjourn state* (introduced in Section 4.2.2). We will describe the database's reaction regarding concurrent conflicting transactions for both, locking and validation-based concurrency control.

The second part of our solution – the Commit tree – deals with the identification of invoked sub-transactions. Furthermore, the Commit tree handles partial restarts of a sub-transaction instead of repeating the whole global transactions.

4.2.1 The Blocking State

The database is allowed to switch unilaterally from the blocking state to the non-blocking Adjourn state as long as the vote has not been sent.

Both states, the blocking state and the non-blocking Adjourn state differ in the way how the validation phase for a concurrent transaction is executed, and therefore show a different blocking behavior.

Blocking State for Locking

If a locking-based concurrency control scheme is used and a sub-transaction T_i that is in the blocking state has acquired the set of read locks $RL(T_i)$ and the set of write locks $WL(T_i)$, another transaction T_k is not allowed to acquire a write lock wl with $wl \in RL(T_i)$, and T_k is not allowed to acquire any lock l with $l \in WL(T_i)$. Thus, a concurrent transaction T_k must wait until T_i is committed or aborted, or until T_i has proceeded to the Adjourn state, and thus has unlocked $RL(T_i)$ and $WL(T_i)$.

Blocking State for Validation

While a successfully validated transaction T_v is in the blocking state, the validation of a newer transaction T_n against the older transaction T_v is done by T_n as

described in Section 3.3.2. This means, transaction T_n is validated against T_v with the effect that whenever transaction T_n is in conflict with T_v, T_n is aborted.

4.2.2 The Non-Blocking Adjourn State

A transaction T_v that has successfully finished its read-phase may enter the non-blocking Adjourn state at any time after T_v has sent the result message to the Initiator and before T_v has sent the vote message. However, T_v must migrate from Adjourn state to blocking state before it may send its vote message to the coordinator.

If validation is used, the database must further perform a *second adjourn-specific validation phase* before a transaction is allowed to leave the Adjourn state.

Definition 4.2.1 The Adjourn state of T_v is a state in which the resource manager RM executing T_v waits for the commit coordinator's demand to vote on T_v, but RM does not block the tuples in $WS(T_v) \cup RS(T_v)$, i.e. the tuples written or read by T_v.

In the following, we describe what happens when a concurrent transaction tries perform conflicting accesses to the data contained in $WS(T_v) \cup RS(T_v)$.

Adjourn State for Locking

If locking is used and a transaction T_v, which has acquired the set of read locks $RL(T_v)$ and the set of write locks $WL(T_v)$, enters the Adjourn state, the locks $RL(T_v)$ and $WL(T_v)$ are released. However, when the resource manager grants one or more locks from the set $RL(T_v) \cup WL(T_v)$ to another transaction T_k while T_v is in the Adjourn state, the RM checks whether or not

$$\begin{aligned} WL(T_v) \cap (RL(T_k) \cup WL(T_k)) &= \varnothing \\ \wedge \quad RL(T_v) \cap WL(T_k) &= \varnothing \end{aligned}$$

If this check evaluates to false, there is a conflict between T_v and T_k. Therefore, the RM locally aborts T_v and the RM can either abort all other corresponding (sub-) transactions that belong to T_v, or try a repeated execution of the sub-transaction T_v if T_v is still repeatable.

Adjourn State for Validation

While T_v is in the non-blocking Adjourn state, the validation of a concurrent transaction T_n is done as follows: T_n is validated against all older transactions *except* those being in the Adjourn state when T_n started its validation phase. This means, T_v, which is in the Adjourn state, has no blocking effect on concurrent transactions T_n.

When T_v must leave the Adjourn state, i.e. when the commit coordinator demands a binding vote on the transaction, T_v must be validated again in a second adjourn-specific validation phase. However, the scope of this second validation is different from the first validation phase:

This second validation of a transaction T_v is successful, if and only if the following condition holds for each transaction T_n that has started its validation while T_v has been in the Adjourn state:

$$\begin{aligned} (RS(T_n) \cup WS(T_n)) \cap WS(T_v) &= \varnothing \\ \wedge \quad RS(T_v) \cap WS(T_k) &= \varnothing \end{aligned}$$

When this validation fails, T_v must either be aborted or can be locally restarted.

The reason for this concurrency check is the following: Although T_v entered its validation phase before T_n, i.e. T_v is older, T_n has not been validated against T_v. Since T_n may have already been committed, the validation of T_v against T_n must be either successful, or T_v must be aborted or locally restarted.

Note that the Adjourn state only delays the validation of T_n against T_v and lets T_v validate against T_n instead of T_n against T_v. However, the number of validation tests is exactly the same as with other commit protocols that use backward oriented concurrent validation.

4.2.3 Local Restarts and Re-Use of Sub-Transactions

Whenever a local sub-transaction T_i executing a Web Service W_i is in Adjourn state and must be aborted due to a conflicting access of a concurrent transaction, the database can try to re-execute the Web service W_i as a sub-transaction T_i'. However, as sub-transactions are dynamically generated, T_i' may invoke different Web services than T_i. Since in our transactional model W_i does not return any values but may invoke other Web services W_k with possible result values, a repetition of W_i only results in a repetition of those Web services W_k that W_i has invoked, but a repetition of W_i will not result in a repetition of the Web service that has invoked W_i.

Furthermore, we can optimize the repetition of the Web service W_i that has spawned a sub-transaction T_i that is repeated as T_i' as follows: T_i' does not need to execute a call to Web service W_s with parameters P_s when the same call has also been issued by T_i with exactly the same parameters. In this case, we can re-use the call of T_i to W_s in the repetition of W_i, since W_s itself is responsible for local restarts in case of concurrency conflicts. Since the effects of the execution of W_s will become permanent after the completion of the atomic commit protocol and W_s never returns any value to an ancestor in the invocation tree, W_s only depends on the invocation parameter of W_i. The concrete execution of W_i, however, does not depend on the execution of W_s at all. Thus, whenever W_s is repeated with the same invocation parameters, this repetition does not have any effect on W_i. Therefore, the call of T_i to W_s can be re-used.

To summarize, if W_i must be repeated and the corresponding sub-transaction T_i has invoked W_s with parameter P_s, the repetition of W_i as T_i' can lead to the following possibilities for the calls to other Web services:

1. T_i' must issue a call to W_s with the same parameters P_s as T_i has done. This call does not need to be executed, T_i' can re-use the invocation done by T_i.
2. T_i' must issue a call to W_s with different parameters P_s'. This call must be executed.
3. T_i' does not need to call W_s anymore. Then, W_s can be aborted.
4. T_i' must issue a call to a new Web Service W_t that has not been invoked by T_i. Then, W_t must be treated as every other Web service that belongs to the global transaction.

If the repetition T_i' of a Web service W_i previously executed as T_i calls exactly the same Web services W_k with the same parameters that were used when T_i called W_k, W_i can be locally repeated without having an effect on other Web services.

4.2.4 Entering the Adjourn State

Each database may decide for itself when it enters the Adjourn state. However, we propose to wait for a short delay in order to avoid unnecessary aborts. Our experiments have shown that waiting for the duration of a typical message delay gives the best results.

Figure 4.1 shows an example application of the Adjourn state when validation based concurrency control is used. It shows two resource managers P_i and P_j that try to commit the sub-transactions T_i and T_j, respectively. Both, T_i and T_j, belong

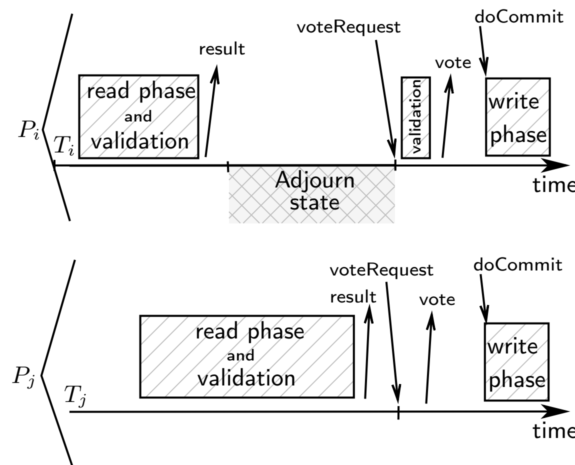

Figure 4.1: Two Sub-Transactions Executed at P_I and P_J

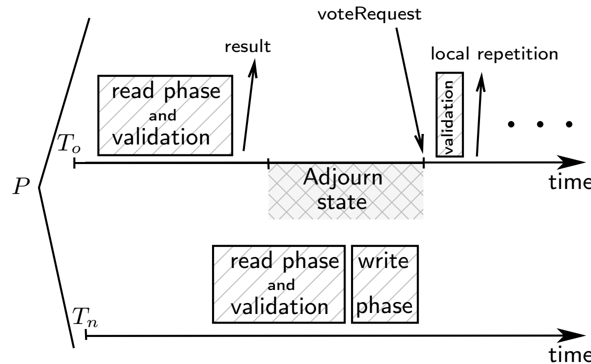

Figure 4.2: Single Resource Manager Running Concurrent and Conflicting Transactions

to the same global transaction T. As the execution time of the read phase and the validation phase takes longer for P_j than for P_i, P_i has to wait for a longer period of time for the voteRequest message to arrive. However, P_i does not know about the delay of P_j. In order to avoid blocking of concurrent transactions after the validation phase, P_i migrates T_i into the Adjourn state and unblocks the occupied resources. After P_j has successfully sent the result, the coordinator demands the vote of P_i and P_j. Since T_i has entered the Adjourn state, P_i must perform the second validation phase as stated in Section 4.2.2, before T_i can leave the Adjourn state and can send the vote to the coordinator.

Figure 4.2 shows a single resource manager P executing the two independent sub-transactions T_o and T_n. While the older transaction, T_o, waits for the coordinator's voteRequest message, the newer concurrent local transaction, T_n, performs conflicting accesses to data tuples accessed by T_o. Therefore, after T_o received the voteRequest, the second adjourn-specific validation phase of T_o against T_n fails, which requires the repetition of T_o's read-phase. However, the delay within the coordination process of T_o has not led to a chain reaction of blocking of the concurrent transactions, e.g. T_n could be still committed.

4.3 Number of Messages

The Adjourn state is entered after the read-phase and the first validation phase have been successfully finished. After the coordinator has sent the voteRequest message, the database performs a second validation, and either immediately replies by sending the vote message, or it locally restarts the sub-transaction. In the failure-free case, each protocol with Adjourn state does not require additional messages compared to the corresponding protocol without Adjourn state. Of course, if a transaction must be locally restarted, additional messages for invoking sub-transactions may be necessary. However, this involves at most the same work and at most the same number of messages as restarting the global transaction as required by protocols without the Adjourn state.

4.4 Commit Tree

As described in Section 4.2.3, a sub-transaction T_i might be restarted as T_i' in case of concurrency conflicts. In this case, T_i and other sub-transactions T_j that have been invoked by T_i and are either invoked by T_i' with different parameters, or are not at all invoked by T_i', can be aborted. In order to abort those sub-transactions, the coordinator must learn about the invocation hierarchy. For this purpose, the invocation hierarchy and the commit status of the involved sub-transactions is stored in a data structure called "Commit tree". To generate the Commit tree, each participant that sends a result to the Initiator attaches the IDs of all invoked sub-transactions. The Initiator then creates the *Commit tree* for a transaction and passes it to the coordinator, which is responsible to maintain the tree in case of restarts.

Each Commit tree belongs to exactly one global transaction and stores the following variables:

1. the global transaction ID,
2. a tree structure containing Commit tree nodes
3. a list unassignedN of unassigned nodes that correspond to sub-transactions T_c that have sent a result before their parent sub-transactions, i.e. the sub-transactions calling those transactions T_c, have sent the result, and
4. a list openSubTransactions of known transaction IDs, for which the result has not yet been received by the Initiator.

Furthermore, each Commit tree node stores

1. the sub-transaction ID of the sub-transactions T_c represented by this node,
2. the ID of the resource manager running the sub-transaction,
3. the transactionID of the parent sub-transaction, and
4. 0 or more IDs of invoked sub-transactions.

When the Initiator has ascertained that the Commit tree is complete, i.e. the Commit tree has an empty list openSubTransactions, it passes the Commit tree to the commit coordinator. Based on the current status of the Commit tree and based on a timer, the coordinator sends the following messages to the participating resource managers:

1. sendVote: when the Commit tree has been received from the Initiator, i.e. all results have arrived at the Initiator and the list openSubTransactions is empty,
2. doCommit when all participants have voted for commit,
3. doAbort: when at least one participant has voted for abort, and
4. doAdjourn: when after a timeout some votes are still missing.

4.4.1 An Example of the Coordinator's Commit Tree

To ensure that all sub-transactions $T_1, \ldots, T_n$ invoked by a sub-transaction T_i are known to the coordinator, the initiator must process the result messages sent by each participant.

Figure 4.3 shows an example Commit tree. Each result message of a sub-transaction T_i includes the result data, the ID of T_i, the ID of the parent sub-transaction

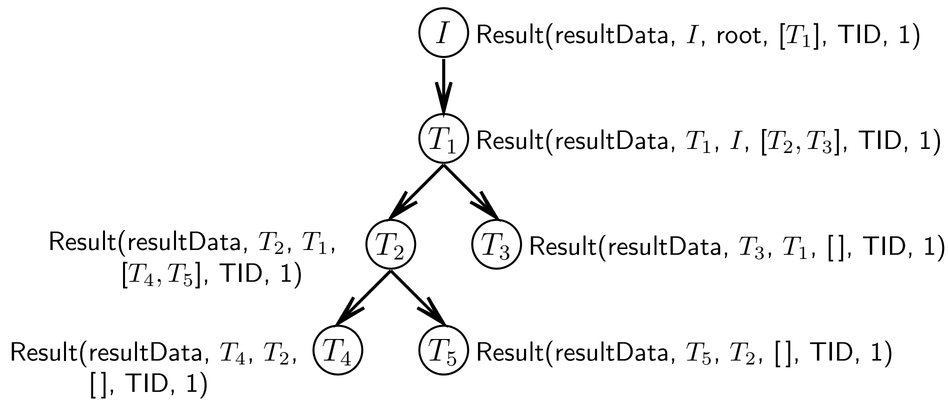

Figure 4.3: Commit Tree Example

that has invoked T_i, a list of sub-transactions that have been invoked by T_i, the global transaction ID TID to which T_i belongs, and a sequence number. When the initiator receives the result of T_1, the node T_1 is created. Since the sub-transaction T_1 has invoked the sub-transactions T_2 and T_3, the vote for commit of the sub-transactions T_2 and T_3 is also required to commit the whole transaction. Therefore, these nodes are added to the commit tree as well. The initiator builds this Commit tree dynamically and determines when all sub-transactions needed for starting the atomic commit protocol execution are finished. Since the information about invoked sub-transactions is sent along with a result message of the parent transaction and the parent's result can be received later than the child's result, it may be the case that a sub-transaction's result cannot be immediately assigned to a node connected in the Commit tree. In this case, the result is stored in a list of unassigned nodes and this node is connected in the Commit tree after the corresponding parent sub-transaction's result has arrived.

4.4.2 Commit Tree Modification by the Result Operation

Whenever a participant has successfully finished its read-phase of a sub-transaction T_i, the following result message is sent to the Initiator in order to invoke the initiator's resultReceived method, which is described in Algorithm 1:

```
resultReceived(Object resultData,
  ID subtransactionID, ID callerID,
  ListOf(ID) invokedSubT,
  ID globalTID, int sequenceNr)
```

The optional parameter “resultData” contains T_i's result, while the “subtransactionID” indicates the ID of T_i. The callerID is the ID of the participant that has invoked T_i. The list “invokedSubT” contains all the sub-transactions invoked by T_i, while the “globalTID” is the ID of the global transaction to which T_i belongs. Furthermore, the result message contains a sequence number, which is increased if the sub-transaction is restarted and a second result message must be sent.

If the sub-transaction was not successful and has been aborted, the participant does not send a “resultReceived” message. Instead, it notifies the Initiator about this abort. Depending on the transaction's implementation, the Initiator might choose a different Web service to fulfill the global transaction.

Algorithm 1 Implementation of resultReceived

```
1: procedure RESULTRECEIVED(Object resultData, ID subtransactionID, ID callerID,
   ListOf(ID) invokedSubT, ID globalTID, int sequenceNr)
2:     if isPreVoteValid(sequenceNr) then
3:         markOutdatedAndAbortUnusedST(subtransactionID, callerID,
                                                  invokedSubT, globalTID);
4:         N :=createNode(subtransactionID, callerID, globalTID, invokedSubT)
5:         openSubTransactions.del(subtransactionID)
6:         if (ParentNode:=getNode(callerID)) == null then
7:             unassignedNodes.add(N) ▷ If parent's result has not arrived yet, put back node
8:         else
9:             ParentNode.addChild(N)
10:            assignNodes(invokedSubT, N)     ▷ Try to assign nodes from unassignedNodes
11:        end if
12:        openSubTransactions.add(invokedSubT)
13:    end if
14: end procedure
```

Algorithm 1 outlines the implementation of the Initiator's resultReceived operation, which is executed on the Commit tree whenever a resultReceived message is received. First, the Initiator uses the sequence number to check that no newer message was processed earlier (line 2). This may be the case when sub-transactions are repeated and certain invocations must be repeated due to different invocation parameters (cf. Section 4.2.3). Therefore, a node N_{old} that represents the sub-transaction subtransactionID may already exist within the Commit tree, but is no longer valid. Thus, the procedure markOutdatedAndAbortUnusedST(...) marks N_{old} as outdated (line 3).

When a sub-transaction is repeated, its invoked sub-transactions may change. To identify sub-transactions that are no longer needed for the commit decision, the IDs of the sub-transactions invoked by N_{old} are compared with the actual invoked sub-transaction parameters invokedSubT of the new result message. Those sub-transactions that are invoked by N_{old} but not needed for commit anymore are aborted and deleted from the Commit tree (line 3).

After this, a new node N is created (line 4) and the parent-child relationships between N and the nodes representing other sub-transactions are managed (line 4-11). In addition, a list openSubTransactions is updated where transactions are stored the votes of which have not yet arrived (line 12).

If all results are present, the list openSubTransactions is empty and the atomic commit protocol, which requires votes for commit of all nodes in the Commit tree, can be started. After the resource managers sent their binding votes, the objects accessed by the transaction are blocked. To ensure that in case of a resource manager failure no infinite blocking of the other resource managers occurs, the coordinator starts a timer. If the time is over and some votes are missing, the coordinator sets the commit status stored in each node of the Commit tree to the Adjourn state, proposes the Adjourn state to T_i and to all sub-transactions belonging to T_i, and demands the votes for the sub-transaction once more.

4.4.3 Commit Tree Modification by Repetition

In case a sub-transaction T_r must be repeated as T_r', the result message with updated parameters must be sent to the Initiator and to the Commit coordinator, and the parameter "sequenceNr" must be increased.

The coordinator replaces the node for T_r with the updated parameters of T_r', and notifies sub-transactions that are not needed anymore, i.e. sub-transactions that were invoked by T_r, but have not been invoked by T_r'. Furthermore, the coordinator demands the votes of those sub-transactions that are additionally invoked by T_r'. Whenever a sub-transaction can be re-used instead of being repeated, the Commit tree does not need to be changed for the re-used sub-transaction.

4.4.4 Benefits of Combining Commit Tree and Adjourn State

The use of the Commit tree in combination with the Adjourn state shows several advantages for transaction processing. As the Commit tree allows to identify all sub-transactions that are invoked during transaction execution, it can be combined

with dynamic transactional models like our Web service transactional model, which allows to invoke sub-transactions even during transaction execution.

In combination with the Adjourn state, the Commit tree allows participants to save energy and to speed up transaction processing for the following reason. Assume, for example, a concurrency conflict has occurred for a sub-transaction T_i. In this case, a repetition of T_i as T_i' including all of its invoked sub-transactions is necessary. However, if during the execution of T_i' some sub-transaction invocations are equal to those performed by T_i, we can re-use the invocations done by T_i and do not need to re-invoke them. This results in time and energy savings.

Furthermore, the Commit tree allows the transaction coordinator to identify and abort sub-transactions that are not needed anymore, for example if sub-transactions have been invoked by T_i but not by T_i'.

4.5 Experimental Evaluation

We evaluated transaction processing in an unreliable mobile environment, which means, participants often disconnect for a short time and come back or, equivalent to this, a lot of messages are lost. Within such a scenario, the longer the blocking of a transaction is (blocking in terms of preventing other transactions from being committed) the greater the risk that another participant will disconnect and does not receive the voteRequest message. Therefore, it is more frequent that the coordinator cannot decide for commit, and that the coordinator must abort the transaction after a time-out than in traditional protocols. In our experiments, we especially focus on two parameters that influence transaction execution and that are characteristic for a mobile network: disconnections and message delay. The adjustment of other parameters such as transmitting power or movement models will finally affect these two parameters. Therefore, we identified “disconnection time and length” and “message delay" as the key parameters that influence the transaction execution. In other words, the more unreliable the network is, the more disconnections, message delays, and message losses occur.

For the simulation, we define a set of scenarios, which differ in both the network events such as disconnections and message delays and the spawned transactions. For each scenario, we start simulating transaction processing at the time when the global transaction is started, and observe the transaction execution until the time when the atomic commit protocol is invoked, i.e. when the coordinator demands the vote. In order to be able to compare the blocking state (which requires a

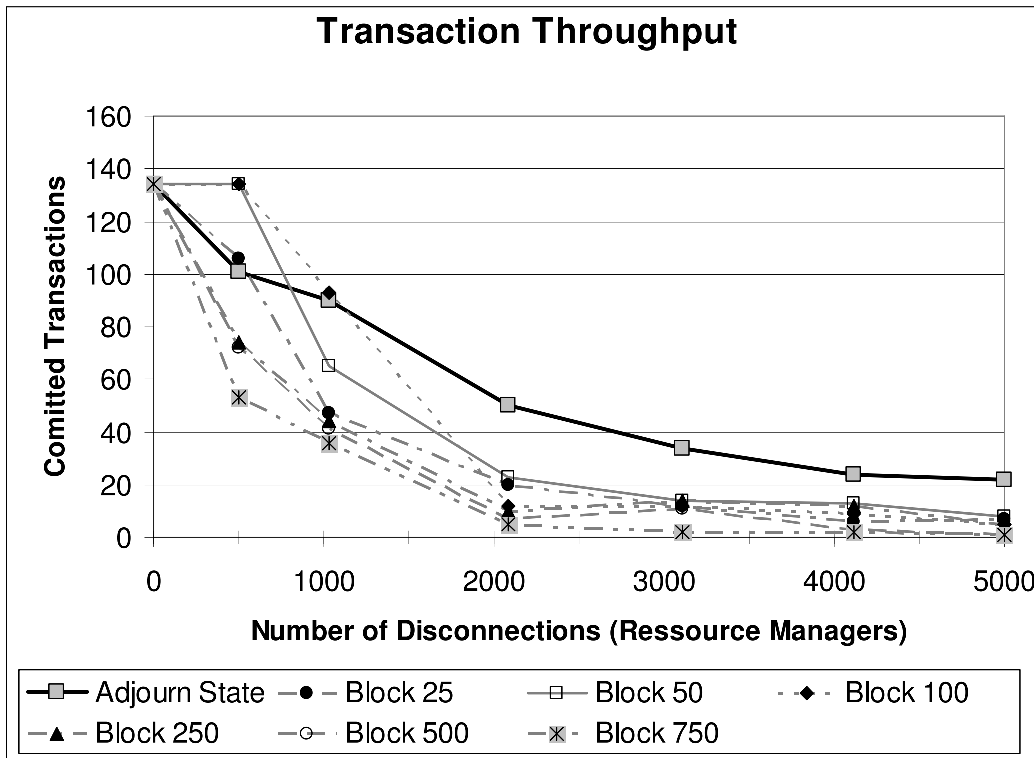

Figure 4.4: Transaction Throughput (Adjourn State and Blocking State)

time-out) and the Adjourn state (which does not require a time-out), we simulate the blocking state with various time-outs. We measure the number of successful transaction executions and the overall blocking time of both the Adjourn state and the blocking state. The concrete parameters of the generated scenario are described in the following subsection.

4.5.1 Scenario

To simulate a varying reliability of our environment, we executed 7 simulations runs, which started when the sub-transaction was sent to each resource manager. The used message delivery is assumed to be fast, i.e. between 0.2 and 2 time units. Each of these 7 runs are stopped after 1000 time units plus additional 60 time units in order to let the time-out based protocols finish the last sub-transactions.

In each of these runs, we let 200 resources execute the same 135 global transactions. Each of these global transactions consists of 4 to 8 sub-transactions, resulting in 830 sub-transactions in total, where each sub-transaction uses exactly one resource.

Most of the sub-transactions have short read phases (randomly selected between 1 and 5 time units), but some transactions contain long read phases (randomly selected between 4 and 25 time units) that delay the transaction's commit.

The 7 runs differ in the number of disconnections of participants. A disconnected participant cannot communicate with other participants as long as the disconnection lasts. In the first run, we let no participant disconnect. In the second run, we randomly add 1000 disconnections to the participants (exponentially distributed). Each of these disconnections has a length of 25 to 50 time units. After 1000 time units, we stop the experiment and count the number of successfully committed transactions.

Predicting an optimal transaction time-out is difficult for the blocking state, thus, we simulated the blocking state using different transaction-time-outs (25, 50, 100, 250, 500, and 750 time units). Each participant starts its time-out after it has sent the result to the coordinator. When the time-out has expired and a participant has not received a voteRequest from the coordinator, the participant aborts the transaction in order to unblock the occupied resources. Furthermore, we measure the total blocking times of all participants.

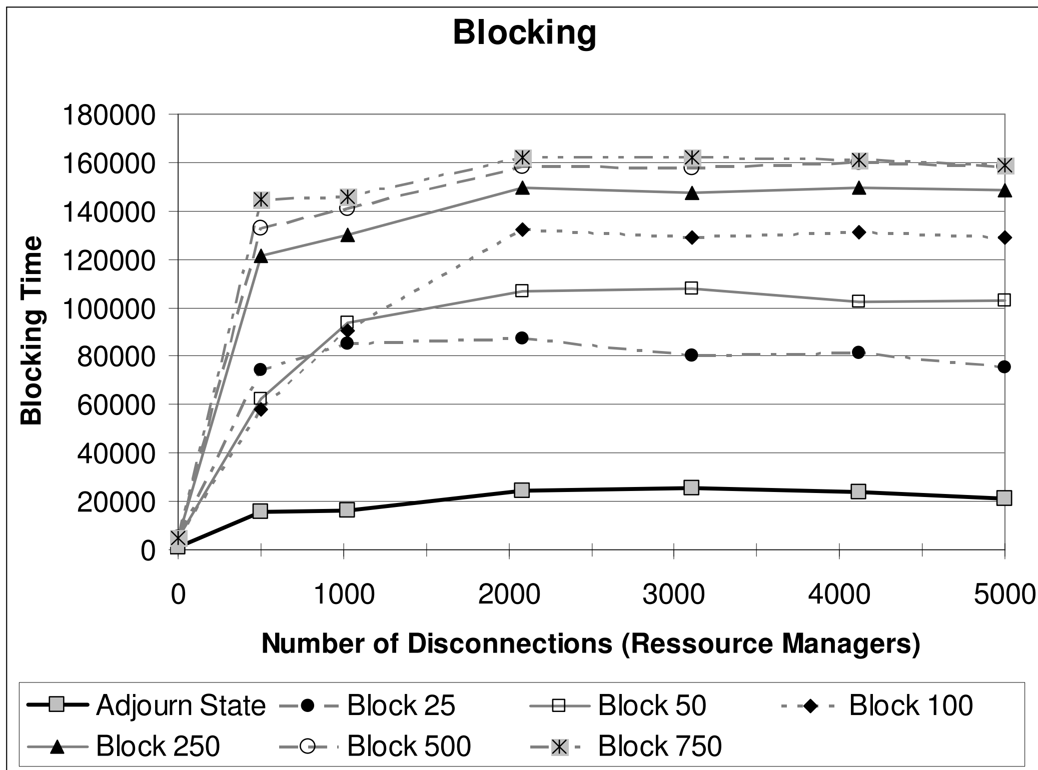

Figure 4.5: Overall Transaction Blocking Time (Adjourn State and Blocking State)

4.5.2 Results

Figure 4.4 shows the transaction throughput for the Adjourn state and for the blocking state when validation is used. On the $x-$axis, the different simulation runs, which vary in the number of resource manager disconnections, are shown. Besides the Adjourn state, Figure 4.4 shows different curves for the blocking state, each of which represents the used transaction time-out. For example, the curve "Block 100" describes a simulation run in which each participant aborts a transaction if the participant has not received the coordinator's voteRequest message for 100 time units after the first validation succeeded. Our experiments confirm that setting up a time-out that maximizes the throughput is difficult and depends on the concrete network reliability. In contrast, the Adjourn state, which does not require such a time-out, shows an average throughput in reliable networks, but is superior to each time-out of the blocking state in unreliable networks.

Besides the transaction throughput, the blocking time is an important criterion for the concurrency control. Figure 4.5 shows the sum of the blocking times of all resources for the Adjourn state and for the blocking state. We can see that the Adjourn state blocks the resources significantly less than each time-out of the blocking state. Again, the overall blocking time of using the blocking state highly depends on the concrete time-out value.

We have conducted similar experiments that use locking-based concurrency control instead of validation. We have relinquished diagrams because these experiments lead to almost identical results. The reason for the similarity of the results is that both concurrency control schemes have the same blocking effect on concurrent conflicting transactions, as explained in Section 3.4.

4.5.3 Evaluation Summary

To summarize, our experimental results have shown that the Adjourn state concurrency control enhancement blocks remarkably less than using the traditional blocking state. Additionally, the Adjourn state achieves a significantly higher transaction throughput in unreliable networks with a lot of disconnections.

Furthermore, our tests have confirmed the difficulty in setting up a database time-out that increases the transaction throughput and reduces the amount of blocking. This justifies the use of the Adjourn state even in mobile networks with moderate reliability, since Adjourn state protocols do not expose the user to the risk of setting up a "wrong" time-out that leads to a performance degradation.

4.6 Related Work

To avoid locking, concurrency control mechanisms like multiversion concurrency control [5, 71], timestamp-based concurrency control [44], or optimistic concurrency control [32, 39] have been proposed. However, these approaches do not solve the problem of setting up time-outs when the database has to abort a transaction. Our proposed Adjourn state does not rely on such time-outs, and merges nicely with these concurrency control mechanisms since it is an "on demand" strategy for giving concurrent transactions access to resources that have been used by transactions which are still waiting for the commit protocol to be invoked.

Compared to the "WS-Atomic-Transaction" proposal [14] our contribution differs in several aspects. For example, [14] has a "completion protocol" for registering at the coordinator, but does not propose a non-blocking state – like our Adjourn state – to unblock transaction participants while waiting for other participants' votes. In addition, our Adjourn state may even be entered repeatedly during the protocol's execution.

[60] proposes 2PC optimizations, e.g. heuristics for committing transactions when messages are lost. However, inconsistencies may occur in case of network partitioning, for example, when some databases do not immediately receive the compensation decision or when the coordination process fails. Furthermore, the approach involves the difficulty of setting up time-outs as well.

Distributed transactions may also occur in the context of mobile agents (e.g. [19, 72]). In this context, the execution code is shipped to the resource managers. Our Adjourn State can be adapted to this context as well.

Since the Adjourn state is used in combination with a dynamic transaction model, the commit coordinator must know the participating sub-transactions. An approach that allows the coordinator to keep track of all dynamically invoked sub-transactions is described in [10].

Our approach is based on the same optimistic principle as [2]. However, the Adjourn state differs from [2] as the Adjourn state does not block resources after the read phase's result has been sent.

4.7 Summary and Conclusion

To summarize, Adjourn state transaction processing does not rely on database timeouts and allows to repeat Web services in case of concurrency failures. Fur-

thermore, we have developed an optimization for Web service repetition that, under certain conditions, can re-use Web service calls instead of repeating them.

We have described the Commit tree, which allows the transaction's commit coordinator to keep track of the commit-status of all participating sub-transactions. Furthermore, we have shown how the Commit tree can be used to unblock participants by migrating them back to the Adjourn state if some participants must repeat their Web service.

We have evaluated our proposed scheme experimentally using simulation. Our experiments have proven the difficulty that traditional protocols involve when setting up time-outs for mobile networks with unpredictable reliability. Our experiments have also demonstrated that using the Adjourn state in unreliable environments can lead to an increased transaction throughput of committed transactions of up to 2.5 times compared to the use of traditional database timeouts. Furthermore, the Adjourn state significantly reduces the amount of data blocking.

Chapter 5

Cross Layer Commit Protocol

5.1 Problem Description

Atomic commit protocols (ACPs) as described in Chapter 2 are used to guarantee the atomic execution of distributed transactions. ACPs are invoked by the transaction Initiator after each participating database has finished the transaction's read-phase. During the execution of an ACP, the following kind of blocking can occur to the atomic commit protocol:

Definition 5.1.1 *Atomic commit protocol blocking* occurs, if an arbitrary sequence of failures leads to a situation where the atomic commit protocol instance cannot terminate with a unique commit or abort decision d during the execution of an atomic commit protocol for a transaction T.

Example 5.1.2 *Assume that the coordinator and one database fail in 2PC after all databases have voted for commit, but before a commit decision was sent out by the coordinator. In this situation, the protocol blocks as it cannot give a decision on the transaction's fate, since the remaining databases do not know the vote of the disconnected database.*

If the coordinator is still alive and only databases disconnect, the protocol is not blocked since the coordinator can immediately decide on the transaction (but may wait a certain time first for the failed database to reconnect).

In this chapter, we focus on atomic commit protocol blocking, while we explain and discuss the problem of *transaction blocking* in Chapter 6.

Atomic commit protocols that are used in mobile environments should not block, even when events like device disconnection, message loss, or network partitioning occur. Thus, failure tolerant ACPs, which have been recently proposed, e.g. by [30,40,58], are preferable for mobile ad-hoc networks in contrast to older approaches designed for performance in fixed-wired environments, e.g. 1PC [29], 2PC [28,46],

or several 2PC optimizations [3, 42, 66]. One of the best protocols regarding failure tolerance is Paxos Commit, which uses multiple coordinators and allows even half of them to fail during protocol execution. This failure tolerance is optimal as described in [63]. However, Paxos Consensus involves several problems when it is used in mobile ad-hoc networks:

- Although it uses multiple coordinators, it is a centralized protocol, i.e. a special leader is still necessary. Each participant may decide for itself during protocol execution if and when it becomes a leader. Even though [30] proposed an additional decentralized variant for a faster commit, this variant does not allow the protocol to terminate decentralizedly if a database's vote message is lost or delayed.
- The number of messages increases with the number of leaders, since each message is routed to the special leader.
- As our experiments have shown, its performance is highly dependent on the use of acknowledgement messages (ACKs). Without ACKs, the performance drains significantly.
- It is an application layer protocol: When ACKs are used, which our experiments have motivated, the protocol is not designed to make any use of these ACKs, i.e., it does not use them for gaining global knowledge.

In this chapter, we present the distributed *Cross Layer Commit Protocol (CLCP)*, which uses multiple coordinators and makes use of acknowledgement messages to piggyback information.

Our CLCP consists of two phases. In the first phase, the *decentralized commit phase*, the participants vote and concurrently try to come to a decentralized commit decision. If a database does not vote for commit at all, CLCP can also abort the transaction within the decentralized commit phase without requiring a centralized leader – in contrast to [30], which needs one or more of such leaders.

However, in the seldom case that the protocol cannot progress due to network partitioning, a *termination phase* will follow. As in [30], one participant becomes a special participant called *leader* that organizes the commit decision and ensures that a majority of participants, i.e. more than 50% of all transaction participants, accept this decision. If the leader fails or the commit decision cannot be made after a timeout, a different participant becomes a new leader having an increased version number that identifies it as the new leader.

We will later see in our experiments that in most cases, the commit decision on the transaction is made within the decentralized commit phase. Due to a decentralized

time-out mechanism, CLCP allows even more transactions to terminate within this phase than [30], which results in a better performance and lower energy requirement of CLCP.

As shown in the last chapter, the Initiator can identify all sub-transactions that belong to a global transaction at the end of the read phase. We assume that before the atomic commit protocol starts, this knowledge is sent to each participant P_i in addition to a request to vote on the transaction.

5.2 Decentralized Commit Phase

5.2.1 Design Goals

The decentralized commit phase was designed to achieve the following goals:

- When knowledge about commit votes is missing for a long time and therefore no commit decision is possible, an agreement to come to an abort decision may be preferable over waiting longer and being blocked.

- Commit coordination is done in a decentralized manner, i.e. without a central leader. Instead, each participant comes to a transaction decision autonomously.

- Avoid blocking the majority of participants when a minority becomes non-reachable by the majority due to sudden network separation during the execution of the commit protocol. Instead, the majority that can communicate shall be conducted to a transaction decision.

5.2.2 Key Design Concepts

The decentralized commit phase is based on the following key design concepts:

- Individual knowledge of a participant $P_i \in P_1 \ldots P_n$ that all participants $P_1 \ldots P_n$ have voted for commit is not sufficient for P_i to commit the transaction for the following reason: The disconnection of P_i can lead to the loss of P_i's vote. When the remaining participants do not have knowledge about P_i's vote, they want to abort the transaction instead of waiting. However, as they do not know whether P_i has committed the transaction, they are forced to wait, which violates the first and the third design goal.

- For this reason, CLCP is not only based on votes, but above all on the *knowledge* that participants have about other participant's votes. Thus, a key aspect of CLCP is that the knowledge of commit votes is exchanged by using a data structure called *commit matrix*. These commit matrices are used for learning the knowledge that other participants have.

- Lost messages or network partitioning may lead to a situation where a participant P_k does not receive the vote of another participant P_v for a long time. This triggers participant P_k to set its knowledge of P_v's transaction decision to the value voteTimeOut, which may, after some messages exchanges, lead to the value timeOutAck and to an abort decision.

- To handle situations where some participants P_k have set their knowledge on P_v's decision to voteTimeOut and other participants P_i have the knowledge that P_v' decision is voteCommit, we use an algorithm that considers the knowledge that all reachable participants have about the votes of other participants. As a commit decision requires majorities, it prevents that another majority exists for abort and vice versa. More precisely, the following is required for a commit decision and for an abort decision.

  1. A commit decision of P_i requires that P_i can be sure that for every participant P_v a majority of participants knows that P_v votes for commit.
  2. An abort decision of P_i due to timeout requires that P_i can be sure that for at least one participant P_v a majority has the knowledge timeOutAck.

  Both, 1. and 2. can never be true at the same time because both rely on majorities.

5.2.3 Commit Matrix

Before we give the decision rules for the first decentralized commit phase, we will explain the *commit matrix*, which plays a major role. The commit matrix is structured as follows:

Each column k of the commit matrix stores the knowledge of one participant P_k regarding the commit votes of all participants. Each row v represents the vote of a participant P_v. An entry (P_v, P_k) describes the entry in row v and column k. For example, an entry $((P_2, P_3) =$ voteCommit) in the commit matrix means that participant P_2 voted for commit, and participant P_3 knows this vote. The entry $(P_2, P_2) =$ voteCommit represents P_2's knowledge of its own voteCommit.

known by / vote of	P_1	P_2	P_3	...	P_n
P_1					
P_2		voteCommit	voteCommit		
P_3					
...					
P_n					

Table 5.1: Example commit matrix

The following entries, ordered with ascending priority, are possible within a cell of the commit matrix: empty $<$ voteCommit $<$ voteTimeOut $<$ timeOutAck $<$ Abort.

An entry (P_v, P_k)=voteTimeOut means that P_k did not receive P_v's vote and thus tries to abort the transaction by timeout. The entry (P_v, P_k)=timeOutAck means that P_k has observed that a majority M of participants $P_i \in M$ has set their knowledge of participant P_v's vote to voteTimeOut or higher, i.e. timeOutAck or Abort. Note, that an abort due to timeout requires two-stages, first, a majority for at least voteTimeOut and thereafter, a majority for at least timeOutAck.

5.2.4 Merging Commit Matrices

During the first phase, participants exchange their own votes and their knowledge of the votes of other participants in terms of their commit matrix. Merging commit matrices is used to learn from other participants and to make sure that the own vote on the transaction is received by a majority of participants. Thus, whenever a participant P_i receives a commit matrix CM_r from participant P_r, the commit matrix CM_r represents the knowledge of P_r. The merge algorithm will adapt these parts of the knowledge of P_r that is not known by P_i or has a higher priority. Thus, whenever participant P_i receives a commit matrix CM_r of P_r, P_i adds all non-empty entries of CM_r to its own commit matrix CM_i, as stated in Algorithm 2.

For all commit matrix entries, P_i checks whether the entry in its own matrix CM_i is equal to voteCommit. If this is not the case and the received commit matrix CM_r contains an entry at the same position with higher priority[1], P_i copies this value into CM_i (lines 4 to 6).

Furthermore, P_i itself can learn new values, thus P_i stores these values in the cells representing the knowledge of P_i, i.e. $CM_i(x, i)$. For this purpose, P_i checks whether an entry $CM_r(x, y)$ with higher priority exists within the corresponding

[1] empty $<$ voteCommit $<$ voteTimeOut $<$ timeOutAck $<$ Abort

Algorithm 2 Merge Commit Matrix (for P_i)

```
1:  local variable: P_i's commit matrix CM_i of size n × n
2:  procedure MERGEMATRIX(CM_r)                          ▷ CM_r is received from P_r
3:      for x, y = 1 ... n do
4:          if (CM_i(x,y) ≠ voteCommit) ∧ (CM_i(x,y) < CM_r(x,y)) then
5:              CM_i(x,y) := CM_r(x,y)
6:          end if
7:          if (CM_i(x,i) ≠ voteCommit) ∧ (CM_i(x,i) < CM_r(x,y)) then
8:              CM_i(x,i) := CM_r(x,y)                    ▷ learn proposal
9:          end if
10:     end for
11: end procedure
```

row x of CM_r(line 7 to 8). If this is the case, P_i learns this newly received value of P_x's commit status by assigning it to $CM_i(x,i)$.

Note that during the first phase, an existing voteCommit entry in the matrix CM_i cannot be changed anymore.

Example 5.2.1 *Assume P_1 has stored the following commit matrix CM_1:*

CM_1	1	2
1	voteCommit	empty
2	empty	empty

Then, P_1 receives the matrix CM_2 of P_2, whose timeout for P_1 has already been triggered, e.g. because P_2 has not received P_1's commit matrix within a certain time.

CM_2	1	2
1	empty	voteTimeOut
2	empty	voteCommit

P_1 merges the matrices as follows: Since $CM_1(1,2)$ is empty and the received entry voteTimeOut of $CM_2(1,2)$ has a higher priority, the received value is copied (Algorithm 2, line 4 to 6). However, $CM_1(1,1)$ is not changed. $CM_1(2,2)$ is directly copied from $CM_2(2,2)$, and $CM_1(2,1)$ is learned from $CM_2(2,2)$ (Algorithm 2, line 7 - 8). Thus, the resulting matrix is:

CM_1	1	2
1	voteCommit	voteTimeOut
2	voteCommit	voteCommit

In this case, the decentralized commit phase described in the following section cannot come to a decision, and thus the termination phase will be started and come to the decision Commit*.*

5.2.5 Decentralized Commit Phase Algorithm

Both phases of our commit protocol, the decentralized commit phase and the termination phase, which is described in the next section, are based on majorities for the commit votes: Whenever a majority has knowledge of a commit vote of a participant P_i, this vote of P_i becomes valid and is anchored within the network. Whenever the votes of all participants are voteCommit and all of these votes are anchored within the network, i.e. each commit vote is known by a majority of participants, the decision to commit the transaction is implicitly made and cannot change anymore, e.g. due to a timeout.

We start by explaining the failure-free case called "decentralized commit phase", which Algorithm 3 summarizes for each participant P_i.

First, the own commit decision knowledge including the vote of participant P_i is broadcasted in terms of the commit matrix CM_i (line 3). P_i then calls a method that waits for and returns the next event, which is either the reception of another commit matrix CM_r, the reception of the transaction's decision from another participant, the reception of a termination vector, or an event timedOut that occurs if the transaction's decision cannot be derived after a predefined amount of time. If another commit matrix CM_r has been received, this matrix is merged with CM_i (line 8) by Algorithm 2. If CM_i has been changed during the merge operation (line 9), the decision rules of Algorithm 4 to Algorithm 7 are executed (line 10 - 15). Then, the matrix CM_i, which was changed by the merge operation and probably by the decision rules, is broadcasted (line 16) as an acknowledgement for the reception of the matrix CM_r. The steps in line 4 to 19 are repeated until a decision (Commit or Abort) is made, a termination vector has been received, or a timedOut event has occurred (line 19). Note that within the same execution of the repeat-until loop, CM_i is broadcasted as an acknowledgement for the reception of CM_r (line 16).

When the decision Commit or Abort has been made, the commit matrix is broadcasted again (line 21) and the thread replyOnRequest is started (line 22), which replies the transaction's decision (including the commit matrix) whenever further commit matrices or termination vectors are received. The thread terminates whenever all participants know the transaction decision. If a termination vector has

Algorithm 3 Decentralized Commit Phase (for P_i)

```
1:  procedure DECENTRALIZEDCOMMIT(CM_i)
2:      Decision := unkown
3:      broadcast(CM_i)
4:      repeat
5:          (CM_r, terminationVector, timedOut, Decision):= waitForNextEvent()
6:          if CM_r ≠ null and Decision = unknown then
7:              CM'_i:= CM_i
8:              CM_i:= MERGEMATRIX(CM_r)
9:              if CM_i ≠ CM'_i then
10:                 COMMITDECISIONRULE(CM_i, Decision)
11:                 if Decision = unknown then
12:                     TIMEOUTATTEMPTDECISIONRULE(CM_i)
13:                     ABORTATTEMPTDECISIONRULE(CM_i)
14:                     ABORTDECISIONRULE(CM_i, Decision)
15:                 end if
16:                 broadcast(CM_i)
17:             end if
18:         end if
19:     until Decision ≠ unknown or terminationVector ≠ null or timedOut ≠ null
20:     if Decision ≠ unknown then
21:         broadcast(CM_i, Decision)
22:         replyOnRequest.startThread(CM_i, Decision)        ▷ Reply if matrix is requested
23:     else
24:         TERMINATIONALGORITHM(terminationVector, CM_i)     ▷ Algorithm 8
25:     end if
26: end procedure
```

been received instead of a commit matrix or a timeout has occurred (line 5), the termination algorithm is executed (line 24).

The following decision rules are checked each time a commit matrix has been received and merged with changes:

Commit

Algorithm 4 shows P_i's decision rule for committing the transaction T involving n participants. It checks whether each row in P_i's commit matrix has a majority of commit votes.

Algorithm 4 Commit Decision Rule (for P_i)

1: **procedure** COMMITDECISIONRULE(CM_i, Decision)
2:     **if** $\forall v = 1 \ldots n$: $\sum_{k=1..n} (CM_i(P_v, P_k) =$ voteCommit$) > \frac{1}{2}n$ **then**
3:         Decision := Commit
4:     **end if**
5: **end procedure**

This commit rule expresses that a participant P_i commits T, if its commit matrix contains the information that each participant P_v voted for commit, and that for each participant P_v, a majority of participants P_k know the vote of P_v.

Timeout Attempt

Algorithm 5 shows P_i's decision rule for assigning the value voteTimeOut to its own Commit Matrix entry $CM_i(P_v, P_i)$ for participants P_v.

Algorithm 5 Timeout Attempt Decision Rule (for P_i)

1: **procedure** TIMEOUTATTEMPTDECISIONRULE(CM_i)
2:     **if** ParticipantVoteTimeout triggered **then**
3:         **for** $v := 1 \ldots n$ **do**
4:             **if** $CM_i(P_v, P_i) =$ empty **then**
5:                 $CM_i(P_v, P_i) :=$ voteTimeOut
6:             **end if**
7:         **end for**
8:     **end if**
9: **end procedure**

This rule expresses that whenever a participant P_i has no knowledge about a decision of a participant P_v after a timeout, P_i sets its own knowledge for P_v to voteTimeOut.

Abort Attempt Due to Timeout

P_i's decision rule for assigning its own Commit Matrix entry $CM_i(P_v, P_i)$ the value timeOutAck is shown by the following Algorithm 6.

Algorithm 6 Abort Attempt Due to Timeout Decision Rule

1: **procedure** AbortAttemptDecisionRule(CM_i)
2: **for** $v := 1 \ldots n$ **do**
3: **if** $\sum_{k=1..n} (CM_i(P_v, P_k) \geq \text{voteTimeOut}) > \frac{1}{2}n$ **then**
4: $CM_i(P_v, P_i) :=$ timeOutAck
5: **end if**
6: **end for**
7: **end procedure**

This abort attempt rule expresses that whenever a majority of voteTimeOut entries or entries with a higher priority, i.e. timeOutAck, exists for a participant P_v, each participant P_i that observes this majority in its commit matrix sets its own entry to timeOutAck.

Abort

Algorithm 7 shows P_i's decision rule for aborting the transaction T involving n participants. This abort rule expresses that whenever either a participant P_v has initially voted for Abort, i.e. the matrix CM_i contains an entry Abort (line 3), or whenever the participant P_i observes that for a participant P_v a majority of timeOutAck entries exist in P_i's commit matrix, P_i aborts the transaction.

Algorithm 7 Abort Decision Rule

1: **procedure** AbortDecisionRule(CM_i, Decision)
2: **for** $v := 1 \ldots n$ **do**
3: **if** $(CM_i\ (P_v, P_v) = \text{abort})\ \vee$
4: $\sum_{k=1..n}\ (CM_i(P_v, P_k) = \text{timeOutAck}) > \frac{1}{2}n$ **then**
5: Decision := Abort
6: **end if**
7: **end for**
8: **end procedure**

Example 5.2.2 *Assume P_1 has gained the following commit matrix CM_1:*

CM_1	1	2	3
1	voteCommit	voteCommit	empty
2	voteCommit	voteCommit	voteTimeOut
3	voteTimeOut	voteCommit	voteCommit

Then, the commit decision rule triggers and P_1 commits the transaction immediately, since a majority of voteCommit *entries exists in each row.*

Assume the following commit matrix for a different transaction:

CM_1	1	2	3
1	voteCommit	empty	voteCommit
2	voteTimeOut	empty	voteTimeOut
3	voteCommit	voteCommit	voteCommit

In this case, the "Abort Attempt Due to Timeout" rule triggers for row 2. Thus, P_1 sets its entry $CM_1(2,1)$, which is voteTimeOut*, to* timeOutAck *and broadcasts the matrix. When P_3 receives this matrix, it also sets its entry $CM_3(2,3)$ to* timeOutAck*, resulting in a majority of* timeOutAck *entries in line 2. Thus, P_3 can immediately abort the transaction and broadcast its commit matrix CM_3.*

5.3 Termination Phase

Due to network partitioning, there may be two cases in which the rule set does not come to a commit or abort/ timeout decision: (a) when no majority of participants exists in one partition or (b) when no majority for commit or abort can be found in the commit matrix after a certain time. Situation (a) is proven to be blocking, according to [63]. For situation (b), i.e. where a majority of participants still can communicate, we employ a solution that, after a timeout, guides this majority to a unique decision although no majority for commit and no majority for timeOutAck can be found in any of the participants' commit matrices. Our solution is based on version numbers and uses an extension of the Paxos Commit algorithm [30] to identify a participant as a leader that is allowed to form a proposal. However, in contrast to [30], we use a distributed termination algorithm that contains only one centralized step: the determination of a new proposal.

5.3.1 Informal Description of the Termination Phase

For each point in time of the termination phase, a participant P_i can have one of two roles: *leader* or *acceptor*. When the termination phase is started for P_i by receiving a termination vector instead of a commit matrix, P_i becomes an acceptor. When the termination phase is started due to a timeout while P_i waits for the protocol's decentralized commit phase to progress, P_i becomes a leader.

An acceptor informs all participants including the newest leader of its actual commit matrix status. The leader then determines a new termination proposal by means of all available acceptor states. This new proposal becomes immediately valid when a majority of acceptors know this proposal. However, each acceptor will only accept proposals from a leader having the highest version number known by this acceptor. Furthermore, whenever an acceptor times out, it can become a new leader, assigning itself a higher version number than the maximum version number known by this acceptor. However, in order to guarantee that each new leader will come to the same proposal once a majority knows this proposal, a new leader must adapt the proposal of the previous leader with the highest version number.

5.3.2 Termination Algorithm

In the termination phase, our commit matrix is replaced by a single *termination vector* TV_i, which is a single row consisting of n entries of the type $TV_i(x) = \langle$bind, version, (proposal, proposalVersion)$\rangle$. An entry $TV_i(x)$ represents the transaction state of participant P_x that is known to participant P_i. Algorithm 8 shows an overview of the termination phase.

At the beginning of the termination phase, the termination vector TV_i has either been received, or is created containing n entries that are initialized with the value 0 in each field (line 2).

A participant P_i identifies the leader P_x having the highest version number v within the termination vector (line 16 - 25) and starts the Acceptor Algorithm for that leader P_x and that corresponding version number v (line 7).

If such a leader P_x cannot be found and the participant's timeout has run up (line 8), the participant itself becomes a *leader* and starts the Leader Algorithm (line 9).

There are three cases in which the Acceptor Algorithm (Algorithm 9) and Leader Algorithm (Algorithm 10), called in line 7 and 9 respectively, terminate:

Algorithm 8 Termination Algorithm (Overview) for P_i

```
1:  procedure TERMINATIONALGORITHM(TV_i, CM_i)
2:      if TV_i = null then TV_i := new(TerminationVector(n)) end if    ▷ n = #participants
3:      Decision := unknown
4:      repeat
5:          (x,v):=FINDNEWLEADER(TV_i)              ▷ x is new leader ID, v leader version
6:          if x≠ null then
7:              Decision := ACCEPTORALGORITHM(x, v, TV_i)
8:          else if (currentTime > (startTime + timeout(P_i)))and(Decision = unknown) then
9:              Decision := LEADERALGORITHM(TV_i, CM_i)
10:             timeout.increase(P_i)              ▷ Multiplied with a participant specific factor
11:         end if
12:     until Decision≠unknown
13:     broadcast(TV_i)
14:     replyOnRequest.startThread(TV_i, Decision)      ▷ Reply if decision is requested
15: end procedure
16: procedure FINDNEWLEADER(TV_i)          ▷ Checks, if P_i is bound to the highest leader
17:     max := i
18:     for x := 1 ... n do
19:         if TV_i(x).version > TV_i(max).version then
20:             max := x
21:         end if
22:     end for
23:     if max ≠ i then return (max, TV_i(max).version)
24:      else return(null, null) end if
25: end procedure
```

1. The protocol has come to a transaction decision in terms of commit or abort. Then, the decision is broadcasted (line 13) and re-sent on request (line 14, cf. the explanation of line 22 of Algorithm 3 on page 54).
2. Another participant P_x has become a leader and has a higher version number than the current leader.
3. The algorithm is stuck, i.e. the transaction decision has not been made after a certain timeout and P_i does not know about a new leader.

In case 2, P_i has noticed a new leader P_x with a higher version number than currently assigned to P_i. Thus, the Acceptor Algorithm for binding P_i to P_x is started immediately (line 7).

In case 3, if P_i has been an acceptor, it re-executes the repeat-until-loop and becomes a new leader due to timeout (line 8-9). Furthermore, it increments its timeout by multiplying it with a participant specific factor in order to give other participants enough time to become a leader before P_i is allowed to restart its Leader Algorithm. To prevent race conditions where two participants restart at almost the same time, the timeout is multiplied with a participant specific factor.

Algorithm 9, the *Acceptor Algorithm*, is started for a participant P_i if the termination vector TV_i contains an entry indicating a new leader P_x. P_i binds to the new leader by setting its own termination vector entry $TV_i(i)$ to the value $\langle$bind $:= P_x$, version $:= v$, (proposal $:= p$, proposalVersion $:= pv\rangle$ (Algorithm 9, line 3), where proposal p and proposal version pv are derived as follows (Procedure ACCEPTORSTATUS(TV_i, CM_i), Algorithm 9, line 16 - 22):

- Whenever a previous leader has made a proposal (line 17), this proposal and the proposed version number are returned.
- Otherwise, whenever the commit matrix CM_i contains an entry for timeOutAck (line 18), p is set to timeOutAck, and since no proposal of a previous leader has been received, the proposal version is set to 0.
- Otherwise, if each column of CM_i contains at least one voteCommit (line 19), proposal p is set to voteCommit and $pv = 0$.
- Otherwise, at least one column of CM_i does not contain a voteCommit, thus p is set to voteTimeOut and $pv = 0$ (line 20).

After this, P_i broadcasts the updated termination vector (line 5) and waits (line 6 and 30) until either a timeout occurs, it recognizes a new leader, it receives a decision on the transaction, or it receives a termination vector entry $TV_i(x) = \langle P_x, v, (pn, v)\rangle$ (line 6). The termination vector entry indicates a proposal of the new leader P_x.

Algorithm 9 Acceptor Algorithm for P_i

1: **procedure** AcceptorAlgorithm(x, v, TV_i)
2:     (p, pv):= acceptorStatus(TV_i, CM_i)
3:     $TV_i(i) := \langle P_x, v, (p, pv)\rangle$ ▷ Note, that $pv < v$
4:     Decision = unknown
5:     broadcast(TV_i)
6:     **if** waitFor($TV_i(x) = \langle P_x, v, (pn, v)\rangle$, TV_i, Decision) = false **then**
7:         **return** Decision
8:     **end if**
9:     $TV_i(i) := TV_i(x)$ ▷ Accept proposal
10:     broadcast(TV_i)
11:     **if** waitFor($(\sum_{k=1..n} (TV_i\,(k) = \langle P_x, v, (pn, v)\rangle)) > \frac{1}{2}n$, TV_i, Decision) = false **then**
12:         **return** Decision
13:     **end if**
14:     **return** $TV_i(i)$.proposal ▷ Decision equals proposal
15: **end procedure**

16: **procedure** AcceptorStatus(TV_i, CM_i)
17:     **if** $TV_i(i)$.proposal $\neq$ empty **then return** ($TV_i(i)$.proposal, $TV_i(i)$.version)
18:     **else if** $\exists a, b : (CM_i(P_a, P_b)$ = timeOutAck) **then return** (timeOutAck, 0)
19:     **else if** $\forall a : (\exists b : CM_i(P_a, P_b)$ = voteCommit) **then return** (voteCommit, 0)
20:     **else return** (voteTimeOut, 0)
21:     **endif**
22: **end procedure**

▷ Common parts of Acceptor and Leader Algorithm

23: **procedure** waitFor(condition, TV_i, Decision)
24:     **repeat**
25:         (CM_r, TV_r, timedOut, Decision) := waitForNextEvent()
26:         **if** Decision $\neq$ unknown **then return** false **end if**
27:         **if** $TV_r \neq$ null **then**
28:             MergeVectors(TV_r, TV_{ID})
29:         **end if**
30:     **until** timedOut **or** condition **or** findNewLeader(TV_i)$\neq$ null
31:     **return**(evaluate(condition))
32: **end procedure**

33: **procedure** MergeVectors(TV_r, TV_i) ▷ TV_r was received
34:     **for** $x := 1 \ldots n$ **do** ▷ Merge TV_r into TV_i
35:         **if** $TV_r(x)$.version > $TV_i(x)$.version **or** ($TV_r(x)$.version = $TV_i(x)$.version **and** $TV_r(x)$.proposalVersion > $TV_i(x)$.proposalVersion) **then**
36:             $TV_i(x) := TV_r(x)$
37:         **end if**
38:     **end for**
39: **end procedure**

P_i then copies this proposal of P_x into its own termination vector entry $TV_i(i)$ (line 9), and broadcasts the updated termination vector (line 10).

When during the procedure WAITFOR( ) a decision on the transaction is received (line 26), the Acceptor Algorithm is immediately terminated and the transaction's decision is returned (line 7 and line 12).

A participant P_i that receives a termination vector TV_r (line 25) *merges* it with its own termination vector TV_i as follows (line 33 - 39). P_i learns about a new proposal, i.e., it sets $TV_i(x) := TV_r(x)$, whenever the received termination vector $TV_r(x)$ contains an entry associated with a version number higher than the version number of the own termination vector entry $TV_i(x)$, or when the version attributes are equal, but the received termination vector contains a higher proposalVersion.

As in the decentralized commit phase, a proposal *pn* becomes valid after a majority of participants has accepted and stored the proposal *pn*, i.e., waiting for this majority was successful (line 11). At this time, each future leader must adopt the proposal with the highest version number and will come to exactly the same proposal. A participant that notices that a majority has accepted a proposal can immediately adopt this proposal as the decision (line 14) and either commit or abort the transaction, depending on the proposal.

Furthermore, the Acceptor Algorithm terminates when a participant recognizes that a new leader with a higher version number has started, or when the condition for which the participant waits (line 6 or 11) does not evaluate to true after a specified amount of time. In this case, the Acceptor Algorithm is immediately stopped and Algorithm 8 continues. If a new leader with a higher version number has appeared (Algorithm 8, line 5), the Acceptor Algorithm is restarted. Otherwise, when the timeout has been triggered, the participant P_i itself becomes a leader (Algorithm 8, line 9) and starts the Leader Algorithm (Algorithm 10).

In Algorithm 10, P_x becomes a *leader* by assigning itself a version number $v := v' + 1$ that is higher than the highest version number v' of which P_x has knowledge (line 2). P_x then sets its own termination vector entry to $TV_x(x) := \langle \mathsf{bind} := P_x, \mathsf{version} := v, (0, 0)\rangle$ (line 3) and broadcasts TV_x as a request to the participants to bind to P_x (line 4).

As different P_x may assign themselves the same value v, the protocol additionally requires that the leader P_x needs the acceptor states of a majority of participants that have been bound to P_x in order to come up with a proposal. Thus, the leader P_x waits until a majority of $\langle P_x, v, \ldots\rangle$ entries exist in its own termination vector TV_x (line 5, note that the asterisk is used as a wildcard). If P_x does not get a majority,

Algorithm 10 Leader Algorithm for P_x

```
1:  procedure LEADERALGORITHM(TV_x, CM_x)
2:      v := v' + 1                                  ▷ v' = highest number ∈ TV_x
3:      TV_x(x) := ⟨P_x, v, (0,0)⟩
4:      broadcast (TV_x)
5:      if WAITFOR((∑_{k=1..n} (TV_x(k) = ⟨P_x, v, *, *⟩)) > 1/2 n, TV_x, Decision) = false then
6:          return Decision
7:      end if
8:      pn = MAKEPROPOSAL(TV_x, CM_x)
9:      TV_x(x) := ⟨P_x, v, (pn, v)⟩
10:     broadcast (TV_x)
11:     if WAITFOR((∑_{k=1..n} (TV_x(k) = ⟨P_x, v, (pn, v)⟩))) > 1/2 n, TV_x, Decision) = false then
12:         return Decision
13:     end if
14:     return pn                                    ▷ Decision := pn
15: end procedure

16: procedure MAKEPROPOSAL(TV_x, CM_x)
17:     if ∃⟨P_x, v, (p, v')⟩ ∈ TV_x : (p ≠ 0) ∧ (v' ≠ 0) ∧ (∀v'' ∈ TV_x.propVersion: (v' ≥ v'')) then
18:         return p
19:     else if ∃a : (TV_x(a) = timeOutAck) then
20:         return Abort
21:     else if ∃a : (TV_x(a) = voteCommit) then
22:         return Commit
23:     else
24:         return Abort
25:     end if
26: end procedure
```

the execution of the procedure WAITFOR( ), which is described in Algorithm 9, and the Leader Algorithm itself terminate when another leader with a higher version number appears, a timeout has occurred, or a decision on the transaction has been received (Algorithm 10, line 6 and 12).

When a majority exists, the leader P_x, and only P_x, is allowed to come up with a new proposal pn (Algorithm 10, line 8) that is determined by the procedure MAKEPROPOSAL(TV_x, CM_x) (lines 16 - 26) according to the following rules.

- When at least one entry of the termination vector TV_x contains a proposal p of a previous leader (line 17-18), the proposal p having the highest proposal version number v' among all proposal version numbers is chosen.
- Otherwise (line 19-20), when at least one timeOutAck is found within the termination vector, there cannot be a majority for voteCommit, since voteTimeOut can only be changed to timeOutAck when there has been a majority for voteTimeOut. Thus, no participant can have committed the transaction, but some may have aborted it by observing a majority for timeOutAck. Therefore, the new proposal is abort.
- Otherwise (line 21-22), no participant has an entry for timeOutAck. Since a majority has replied to P_x and no entry for timeOutAck was found, there cannot have been a majority for timeOutAck. Thus, the transaction cannot have been aborted because a majority of timeOutAck must be present before a transaction can be aborted. If at least one voteCommit is present, the proposal is commit.
- Otherwise (line 24), no voteCommit is found and the transaction will be aborted.

The new proposal pn of the leader P_x is broadcasted as a termination vector entry $TV_x(x)$ with $TV_x(x) := \langle$bind $:= P_x$, version $:= v$, (proposal $:= pn$, proposalVersion $:= v$)$\rangle$ (line 9 - 10). When sufficiently many participants P_k have accepted this proposal, have stored it into their own termination vector entry $TV_k(k)$, have broadcasted their termination vector TV_k, and P_x has received and merged termination vectors into its own termination vector TV_x such that TV_x shows that a majority of participants accepted P_x's proposal pn, then waiting for this majority (line 11) was successful. Thus, the proposal pn becomes valid and is returned as the transaction's decision (line 14).

To ensure that the protocol continues even when P_x disconnects, another participant P_k may become a new leader and assign itself the version number $v + 1$.

Note that two leaders can come to two different proposals depending on their termination vectors. However, only the one having the highest version number will succeed, since both leaders must bind a majority of participants before they are allowed to make a proposal. Thus, a majority of participants will reject the proposal of the leader with the smaller version number when they are already bound to the higher version number (cf. Algorithm 9).

Example 5.3.1 *Assume four participants P_1 to P_4, each of which has its commit matrix CM_i filled with 2 entries voteCommit and 2 entries voteTimeOut in each row. Thus, the decentralized commit phase is stuck, and after a timeout, the termination algorithm starts. Let P_1 be the first participant whose timeout occurred. Then, P_1 becomes the leader and sets:*

TV_1	1	2	3	4
	$\langle P_1, 1, (0,\ 0)\rangle$			

After P_2 has received TV_1, P_2 binds to the leader P_1 by copying the leader's termination vector entry $TV_1(1)$ to $TV_2(1)$ and derives its own acceptor status (voteCommit) and acceptor version (0) (Algorithm 9, line 19). Therefore, P_2 adds the entry $\langle P_1, 1, (\mathsf{voteCommit}, 0)\rangle$ to its own termination vector and broadcasts the vector.

TV_2	1	2	3	4
	$\langle P_1, 1, (0,\ 0)\rangle$	$\langle P_1, 1, (\mathsf{voteCommit}, 0)\rangle$		

P_3 and P_4 do the same as P_2. When sufficiently many termination vectors have been exchanged such that P_1 has noticed that a majority of entries $\langle P_1, 1, (*, *)\rangle$ exists in the termination vector of P_1, P_1 knows that a majority of participants has bound to the leader P_1. Then, P_1 creates a new proposal. In our example, this proposal is commit since the acceptor status of P_3 and P_4 is voteCommit. P_1's proposal is stored in $TV_1(1)$ as the entry $\langle P_1, 1, (\mathsf{Commit}, 1)\rangle$. The participants recognize that this is a proposal by means of the proposal and proposal version number (Commit,1).

TV_1	1	2	...
	$\langle P_1, 1, (\mathsf{Commit}, 1)\rangle$	$\langle P_1, 1, (\mathsf{voteCommit}, 0)\rangle$	...

When P_2 receives TV_1, it copies the entry $TV_1(1)$ to $TV_2(1)$ and accepts the proposal by changing its own entry $TV_2(2)$ to $\langle P_1, 1, (\mathsf{Commit}, 1)\rangle$:

TV_2	*1*	*2*	...
	$\langle P_1, 1, (\mathsf{Commit}, 1)\rangle$	$\langle P_1, 1, (\mathsf{Commit}, 1)\rangle$	...

When P_3 has received the termination vector of P_2 and has also accepted the proposal, i.e. P_3 has set its entry $TV_3(3) := \langle P_1, 1, (\mathsf{Commit}, 1)\rangle$, it can commit the proposal immediately since it knows that a majority has accepted the proposal of the leader P_1.

5.4 Correctness and Liveness

5.4.1 Correctness

In order to prove correctness, we prove that all participants that come to a transaction decision will come to the same transaction decision and we prove that each participant's transaction decision is stable, i.e. it is not changed by the protocol.

Definition 5.4.1 A participant comes to a *transaction decision*, when its local variable Decision takes a value different from unknown, i.e. commit or abort.

Lemma 5.4.2 *A participant that has come to a transaction decision will never come to a different decision.*

Proof (Sketch) When the local variable Decision is set to a value different from unknown in Algorithm 3 or Algorithm 8, the protocol will never write on the variable Decision again.

Lemma 5.4.3 *All participants of CLCP that come to a commit or abort decision come to the same decision.*

Proof (Sketch) Since the protocol is based on majorities, we first show that whenever a participant has come to a valid Commit decision in the decentralized commit phase (Case C1) or in the termination phase (Case C2), no participant can come to an Abort decision anymore.

Case C1 At least one participant comes to a commit decision in the first phase. A participant can only come to a commit decision whenever, for each participant, a majority of participants P_i has stored the voteCommit in the rows/columns of its commit matrices CM_i. In this case, no commit matrix CM_k can have a majority for voteTimeOut due to the commit rule of Algorithm 4. Thus, due to the abort attempt rule of Algorithm 6, no entry for timeOutAck can exist.

Therefore, due to the abort decision rule of Algorithm 7, no participant can abort the transaction in the first decentralized commit phase.

In order to come to a transaction decision when one or more participants have started the second termination phase, a majority of all participants must reply to a leader with its acceptor states (Algorithm 9, line 16 - 22). Since at least one participant has come to the commit decision by assumption, for each participant a majority of participants P_i has stored voteCommit in the rows/columns of its commit matrices. Thus, no participant can have a timeOutAck in its commit matrix, and there cannot be an acceptor status timeOutAck. Thus, the acceptor status for all P_i is voteCommit (Algorithm 9, line 19). Since each new leader P_l must receive the acceptor states of a majority of participants, P_l will receive at least one voteCommit from one of the participants P_i, but P_l will not receive timeOutAck. Thus, the proposal built by P_l in the termination phase will always be Commit (Algorithm 10, line 16 - 26).

Case C2 A participant P_i has come to a commit decision in the termination phase. In this case, there must have been a majority of entries for commit in the termination vector of P_i associated with a version number v. This is possible after a leader P_x with version number v has succeeded. Any future leader must first bind a majority of participants and then adopt the previous proposal with the highest version number. Let P_k be the first leader after P_x. As P_k requires support of a majority of acceptors, P_k will receive at least one entry in the termination matrix with version number v. Since P_k is the first leader after P_x, v is the highest version number. Thus, P_k must adopt the proposal for commit and cannot come to an abort decision.

In the second part of the proof, we show that whenever a participant has come to an Abort decision, no participant can come to a Commit decision anymore. We distinguish two cases:

Case A1 A participant P_j is not able to commit a transaction, e.g. due to database constraints. Then, P_j immediately aborts the transaction. Since P_j will not have sent a voteCommit, any decision process will lead to abort due to the missing vote of P_j.

Case A2 Whenever a participant has come to a valid abort decision in the first or second phase, the same argumentation as for Case C1 and Case C2 holds: An

abort due to timeout requires a preceding majority for timeOutAck, thus no majority for voteCommit can exist.

5.4.2 Liveness

In the remainder of this section, we consider liveness, i.e. we prove that the protocol will lead a majority of participants that can communicate with each other to a transaction decision, and that this decision is the same for all participants of the majority.

However, if the network is partitioned in such a way that no majority of participants can communicate anymore, there is no way to decide on the transaction without violating the requirements for a unique transaction decision [63].

The following definitions are used for proving liveness of CLCP's termination phase.

Definition 5.4.4 We call a leader P_x with version number v *dead*, if a majority of participants has bound to a leader P_j with version number w and $w > v$, i.e. a majority of participants has set its termination vector entry TV_j to the version number w (Algorithm 9, line 2 - 3).

Definition 5.4.5 We call a proposal p of a leader P_x with version number v *anchored*, if each following leader P_j with version number w, $w > v$ that does not become a dead leader but makes a proposal, will propose p as well.

Note that a proposal p becomes anchored if a majority of participants has accepted and stored p.

Lemma 5.4.6 *CLCP will come to an anchored proposal whenever a majority of participants is in the termination phase and the participants of the majority can communicate with each other for a sufficiently long period of time.*

Proof (Sketch) During the termination phase, a participant P_i that executes Algorithm 9 or Algorithm 10 only waits during the execution of the procedure WAITFOR (described in Algorithm 9, line 23 - line 32) for the following events to occur:

1. A leader condition is fulfilled (Algorithm 10), which is either
   (a) a majority of participants has bound to the leader, i.e. the participants have set their termination vector entry to the version number associated with the leader (Algorithm 10, line 5), or

   (b) a majority of participants has accepted the leader's proposal (Algorithm 10, line 11).

2. An acceptor condition is fulfilled (Algorithm 9), which is either

   (a) the leader has made a proposal (Algorithm 9, line 6), or

   (b) a majority of participants has accepted the leader's proposal (Algorithm 9, line 11).

3. A timeout has occurred.

4. A new leader with a higher version number has appeared.

The appearance of a new leader with higher version number (event 4), can only occur if a participant's timeout has triggered. Thus, we can reduce event 4) to event 3), since both events result in a restart of the leader and acceptor algorithms. In the following, we argue that the time between two occurrences of the events 3) and 4) will grow larger due to the increment of the timeout value after a restart of a new leader, which gives a leader enough time to succeed:

Assume a participant P_i becomes a leader before a previous leader P_x had time to anchor its proposal. The protocol ensures that each leader that has died must increase its participant specific timeout after the termination of the leader algorithm (Algorithm 8, line 10). Thus, the timeouts after which participant P_x decides to become a leader again will grow larger with an increasing length of time. Furthermore, the timeouts are different for each leader since the timeouts are increased by being multiplied with a participant specific factor. At some point in time, the timeout has grown to an extent at which the amount of exchanged messages will ensure that a leader's proposal is anchored before a new leader will become present, more precisely, there is a point in time at which the events 1b) and 2b) will occur before one of the events 3) and 4) triggers.

When none of the events 3) or 4) occur for a sufficiently long period of time, the events 1) and 2) let the protocol progress: Each acceptor broadcasts its status; the leader receives the acceptor states of a majority of acceptors, builds a proposal, and broadcasts the proposal; each acceptor will accept the proposal. Thus, after sufficiently many exchanged messages, the protocol will anchor a proposal.

Thus, after a sufficiently long period of time, a leader exists that has been able to anchor its proposal.

Theorem 5.4.7 *Whenever a majority M of participants can communicate with each other for a sufficiently long period of time, each participant $P_i \in M$ will come to the same transaction decision in terms of commit or abort.*

Proof In the decentralized commit phase, each participant P_i exchanges commit matrices and changes its commit matrix until either the decision rules return a decision on the transaction, or until a previously defined timeout occurs. When a decision on the transaction is made, this decision is broadcasted as an acknowledgement to each participant that does not know about this decision. Each participant that learns about the new decision terminates the protocol and returns the decision to other participants as well. Thus, whenever a single participant of the majority M has derived a decision, this decision will be sent to all other participants of M and the protocol terminates.

When the predefined timeout occurs and no decision could be derived, each participant P_i starts its termination phase. Lemma 5.4.6 states that after a sufficiently long period of time, a proposal becomes anchored and thus becomes the transaction's decision. Lemma 5.4.2 states that a known decision is not changed anymore, while Lemma 5.4.3 states that all participants come to the same decision.

5.5 Experimental Evaluation

We have evaluated our protocol in an environment generated by the BonnMotion mobility scenario generator [70]. We simulated various mobility models, i.e. Random Waypoint, Attraction Point, and Manhattan Geometry. Transactions are issued to participants that are close together initially, but each of the participants may move or disconnect during the commit protocol execution. Furthermore, we assumed a message reception model that covers the real-world reception behavior better than the standard unit-disc-model.

5.5.1 Quasi-Unit-Disc Reception Model

In contrast to the unit-disc-model, we use the *quasi-unit-disc-model* [26] as reception model, which uses multiple probabilities for message reception as follows. Whenever a mobile device T transmits data, a participant R_i that has the distance Δ_i to T receives this data with the probability

$$p = \min\left(1,\ 1 - \frac{\max\left(0,\ \Delta_i - \mathsf{GuaranteedRange}\right)}{\mathsf{MaxRange\text{-}GuaranteedRange}}\right)$$

where MaxRange is the maximum sending range, and GuaranteedRange>0 denotes the range with guaranteed message reception in our model.

Whenever GuaranteedRange $< \Delta_i <$ MaxRange holds, we say that the receiver is located within the *Possible Reception Range (PRR)*. For receivers that are located within the PRR of a sender, our model assumes linear reception probability depending on the actual distance to the sender.

5.5.2 Consequences for Atomic Commit Protocols

As [26] has shown, the results for routing when using the quasi-unit-disc-model differ from when using the unit-disc-model. In mobile networks, a message sender cannot be sure of the delivery of the message, but the successful delivery is crucial for the ACP to progress and to commit the transaction. Thus, acknowledgements (ACKs) are widely used, and the message transfer is repeated when the ACK has not been received by the sender after a certain time. Unfortunately, each acknowledgement requires the intended receiver to additionally transmit data and spend energy. Furthermore, ACKs may get lost, causing the transmission to be unnecessarily repeated. As our experiments show, the use of standard atomic commit protocols without ACKs leads to a low transaction throughput and a high abort rate, especially when devices are often within the PRR. However, having ACKs enabled, the message costs of the used protocols increase significantly, depending on the number of hops for each message. This motivates the development of our cross-layer protocol CLCP that makes use of the ACKs to piggyback information.

5.5.3 Setup

For each of these mobility models, we created N distributed transactions for devices that are close together. We have assumed that during one time unit, there is no limit to the number of sending actions, but the data that is sent is received within the next time unit. Although our experiments include some parameters that are based on chance, we used the random seed of the random number generator to produce repeatable experiments.

Table 5.2 shows the parameters that we used in our evaluation.

These parameters and the following parameters for the mobility model are motivated by the BonnMotion mobility generator [70] and proposed as standard parameters. However, our experiments have shown that there is no significant influence on the overall comparison of the protocols when varying concrete parameters.

Common Parameters for Each Experiment	
Guaranteed transmission range	10
Max transmission range	60
Network timeout	20
Random seed	1001

Table 5.2: Experimental Evaluation Parameters

5.5.4 Mobility Models

We simulated multiple ad-hoc networks based on the following mobility models:

Random Waypoint Model assigns a random destination to each node within a predefined area, towards which the node moves on the shortest route without leaving the predefined area.

Attraction Point Model is based on the Random Waypoint Model, but has predefined destinations towards which the nodes move.

Manhattan Model assigns a random destination to each node at a predefined grid. Thus, the nodes move only on the grid.

We used the parameters shown in Table 5.3 for creating the three different mobility models. For further information regarding the models, we refer to the literature, [15], for example, gives an overview of several mobility models.

5.5.5 Transaction Generation

We generated transactions on sets of participants that are connected by a spanning tree with a maximum edge length of distance d. Thus, we analyzed the generated mobility models and whenever we could identify a predefined number n of participants that form a spanning tree with predefined maximum edge length d, we generated a transaction among these closely connected participants. When the number of generated transactions is greater than the predefined number of transactions, we delete transactions randomly but equally distributed until we have exactly as many transactions as defined. Table 5.4 shows the transaction generation parameters that we used during our experiments.

Our experiments have shown that a variation of the transaction's read-phase duration results in a similar behavior as varying the maximum edge length d of the minimum spanning tree. The reason is that when the ACP starts after the completion of a long lasting read-phase, some nodes have already moved, which results in the same behavior as specifying a greater maximum edge length d with a shorter

Common Mobility Model Parameters	
Number of nodes	100
Simulation duration (in time units)	15,000 TU
Initial cut-off (in time units)	3,600 TU
Number of nodes	100
Field width (in field units)	500 FU
Field height (in field units)	500 FU
Random seed	1,000

Random Waypoint Model	
Min node movement speed per TU	0.5 FU
Max node movement speed per TU	1.5 FU

Attraction Point Model	
Attraction points (x, y, weighting, std. deviation of x/y coordinates)	(467, 463, 7.3, 2), (244, 154, 7.8, 1), (472, 444, 8.6, 14), (205, 239, 6.8, 14), (246, 426, 3.4, 4), (169, 490, 7.0, 25), (340, 322, 1.8, 43), (57, 305, 3.7, 19), (235, 239, 2.6, 5), (28, 315, 6.2, 37)
Random waypoint parameters apply as well.	

Manhattan Model	
Grid layout	5x5

Table 5.3: Parameters of our Mobility Models

Transaction Generation	
Number of Transactions	1000
Maximum spanning tree edge length d	50 FU

Table 5.4: Transaction Spawning Parameters

read phase execution time. Thus, to reduce the number of varying parameters, we assume that the transaction's read-phase is equal for all (sub-) transactions and consists of one time unit, but we varied the number of involved participants during our experiments.

5.5.6 Routing Protocols

In order to be able to compare 2PC, Paxos Commit, and CLCP and to cope with our reception model that allows a message to be simultaneously received by multiple participants, we have implemented a *Relay Routing protocol* for CLCP that involves all transaction participants and additionally some nodes that are not participating within the transaction. Relay Routing uses a kind of bounded flooding with a maximum number of non-transactional participant relays. Thus, we do not rely on pro-active routing components, and we do not require any knowledge of the network at any time.

Whenever a transaction participant P_s must send a message to a transaction participant P_r, each transaction participant P_k that has received this message, i.e. P_k is either within the PRR or within the guaranteedRange, re-sends the message once. Whenever a non-transaction participant N_i receives a message, N_i increases the message counter relay and then also re-sends the message until the counter relay is greater than or equal to a pre-defined maximum relay value maxRelay. Whenever a participant receives the same message a second time, the message is discarded.

Furthermore, we compared CLCP using Relay Routing with 2PC and Paxos Commit using the "Nearest Forward Progress Routing Protocol" (NFR) [33], which lets each sender forward a message to the nearest node that reduces the distance to the message receiver. For these experiments, we assumed a global knowledge of all other node positions. Further details on these routing protocols can be found in [27]. The parameters that we used in our evaluation are shown in Table 5.5.

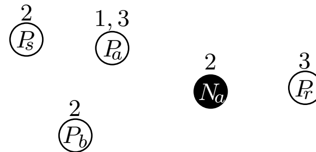

Figure 5.1: Relay Routing Example

Example 5.5.1 *Figure 5.1 shows a routing from P_s to P_r. The numbers above the nodes indicate the logical time at which the message is received. After P_s has sent the message at time 0, P_a receives the message at time 1, but P_b does not. P_a then re-sends the message, which is received at time 2 by P_b, P_s, and the non-transaction participant N_a. Since P_s has already sent the message, P_s discards it. P_b as well as the non-transaction participant N_a, re-send the message, assuming a maximum relay value maxRelay greater than 0. At time 3, P_a and P_r receive the message and*

Relay Routing without ACK	
Simulated ACPs	CLCP, 2PC, Paxos Commit
maxRelay (non transaction participants)	1
Positioning information	None
Max hop count	1
Use message ACK	Yes, no

Nearest Forward Progress Routing	
Simulated ACPs	2PC, Paxos Commit
Positioning information	Global knowledge
Max hops	8
Max message retry	1
Max hop count	10
Use message ACK	Yes

Table 5.5: Routing Protocol Parameters

the routing terminates. If a broadcast among the transaction participants must be issued, no recipient for the message would be given. Then, P_r must re-send the message at time 4.

With each routing protocol, there is no guarantee that the message will arrive at its destination. Thus, a participant P_s could require P_r to send an acknowledgement ACK when P_r has received the message. Whenever P_s does not receive ACK after a certain time, P_s must re-send the whole message again. As 2PC and Paxos Commit require messages to be sent to a specified destination, we have simulated both atomic commit protocols twice, i.e. each with ACKs toggled on and off. CLCP, in contrast, only requires "broadcasts" and does not need special ACKs since each participant acknowledges a received Commit Matrix/Termination Vector by sending the updated matrix/vector in the next broadcast message.

5.5.7 Energy Consumption

In our experiments, we measured the totally used energy that was needed for sending and receiving packets using the following terms, which were given by [25]

due to measurements for a Lucent IEEE 802.11 2 Mbps Wavelan PC-Card. For broadcasts, the following energy computation was used:

$$\text{send packet of } n \text{ bytes: } 1.9\mu W{\cdot}sec \cdot n + 266\mu W{\cdot}sec$$
$$\text{receive packet of } n \text{ bytes: } 0.5\mu W{\cdot}sec \cdot n + 56\mu W{\cdot}sec$$

For point-to-point routing, the following energy computation, whose higher fixed cost is associated with the IEEE 802.11 control protocol, was used, corresponding to the measurements of [25].

$$\text{send packet of } n \text{ bytes: } 1.9\mu W{\cdot}sec \cdot n + 454\mu W{\cdot}sec$$
$$\text{receive packet of } n \text{ bytes: } 0.5\mu W{\cdot}sec \cdot n + 356\mu W{\cdot}sec$$

5.5.8 Results

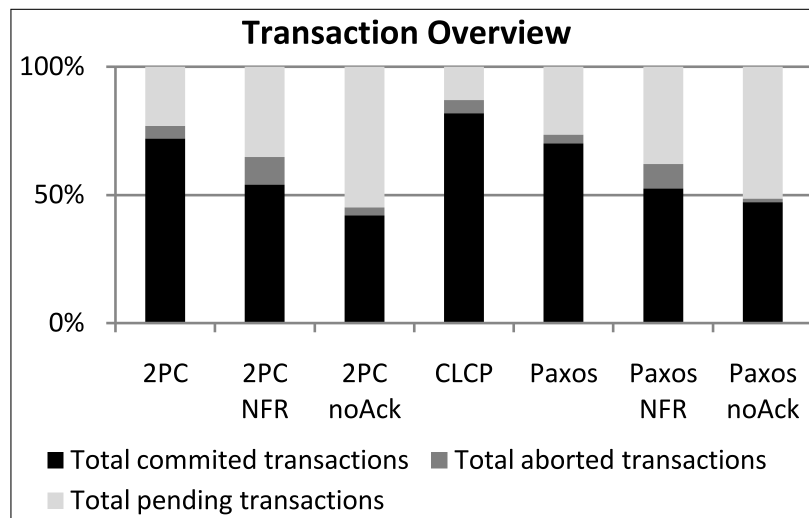

Figure 5.2: Manhattan Mobility Model, 5 Participants

Figure 5.2, 5.3, and 5.4 show the percentage of committed transactions, the percentage of transactions that have been aborted due to timeout, and the percentage of pending transactions for each protocol involving 5 transaction participants. We defined a *pending transaction* as a transaction, for which at least one participant has not received the transaction decision, i.e. Commit or Abort. A low amount of pending transactions is one of the main criteria for actually using a particular commit protocol in practice, since pending transactions block resources and thus prevent the database from executing concurrent transactions that conflict with these pending transactions. However, as the number of pending transactions could be reduced

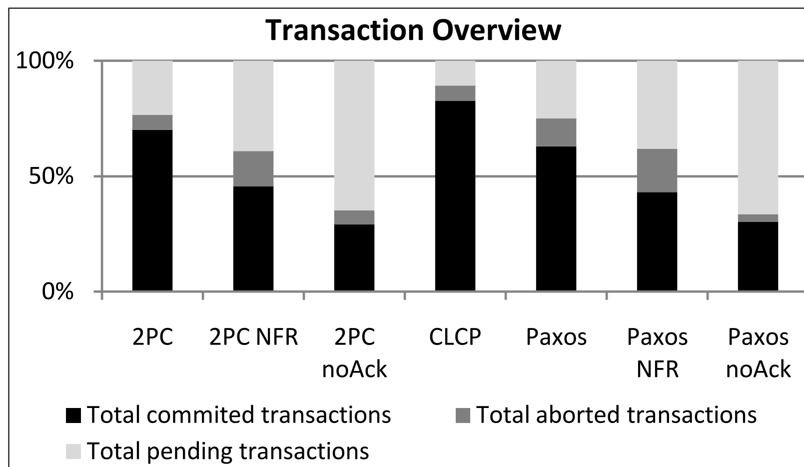

Figure 5.3: Random Waypoint Mobility Model, 5 Participants

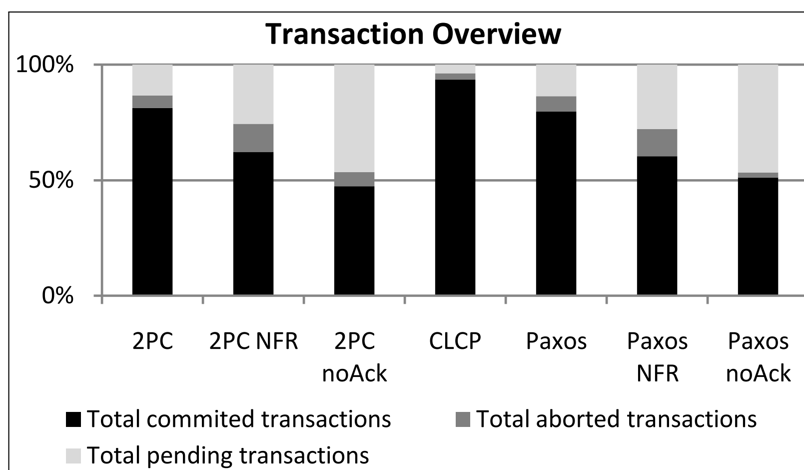

Figure 5.4: Attraction Point Mobility Model, 5 Participants

by a protocol that aborts each transaction, the reduction of pending transactions and the increase of committed transactions will indicate a protocol's quality.

"2PC NFR" and "Paxos NFR" both use the "Nearest Forward Progress" routing, while all other protocol implementations use Relay Routing as described in Section 5.5.6. Both, "2PC noAck" and "Paxos noAck" omit acknowledgement messages. Thus, participants using a noAck protocol send each message only once, regardless of whether it is received or not.

As we can see, CLCP is able to commit more transactions than all other protocols, and CLCP reduces the number of pending transactions in each mobility model. The protocols using NFR show a significantly worse behavior regarding the number of

committed transactions, motivating the use of Relay Routing for the atomic commit protocol.

The same applies to protocols that do not use acknowledgement messages. As expected, the number of pending transactions is significantly larger for 2PC and for Paxos Commit when not using acknowledgements.

Figure 5.5, 5.6, and 5.7 show the total network energy consumption in $kW * sec$ depending on the number of transaction participants, but regardless of the number of committed transactions. Note that we scaled the y-axis logarithmically and measured the required energy for all protocols including the energy that is required for the exchange of CLCP's commit matrices.

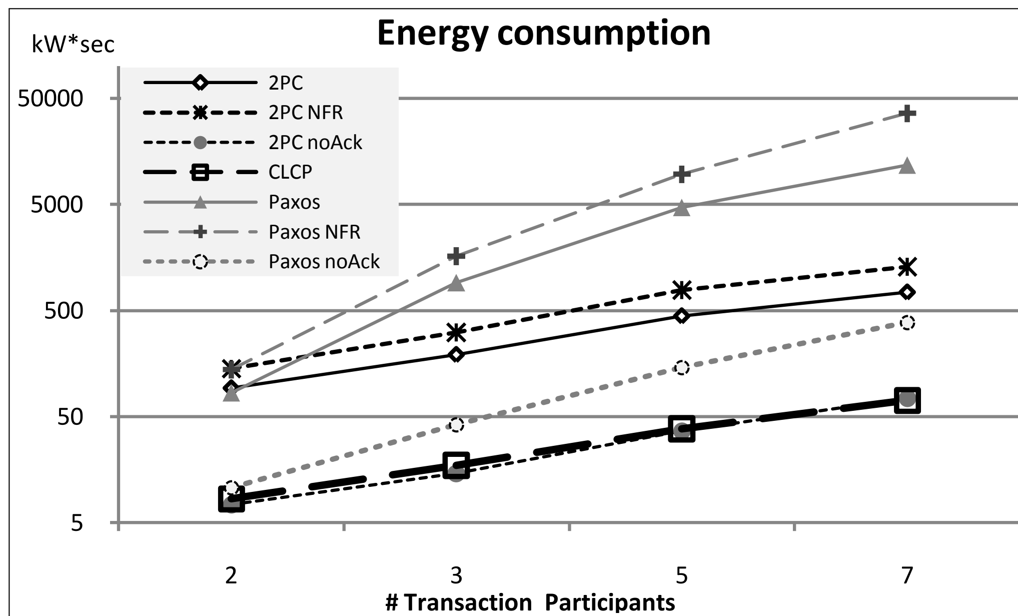

Figure 5.5: Manhattan Mobility Model

As expected, protocols that do not use acknowledgement messages need significantly less energy than similar protocols using them. In contrast, CLCP needs almost the same low amount of energy as “2PC noAck”, but – as we have seen in Figures 5.2, 5.3, and 5.4 – shows the highest number of committed transactions. “Paxos NFR” and “Paxos” turned out to be very expensive in terms of energy, while the “Paxos noAck” needs remarkably less energy (but shows a poor number of commits).

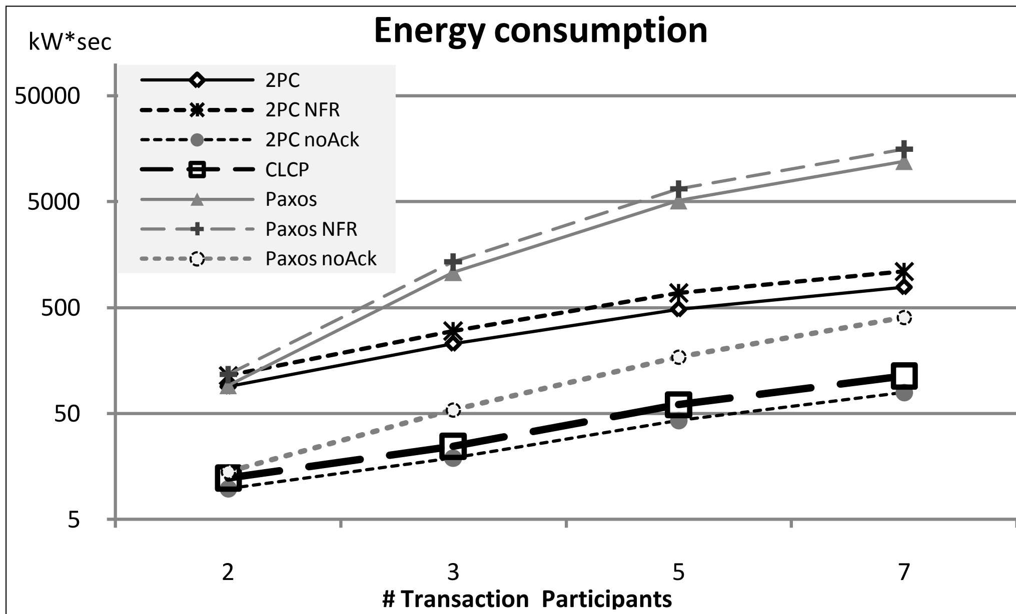

Figure 5.6: Random Waypoint Mobility Model

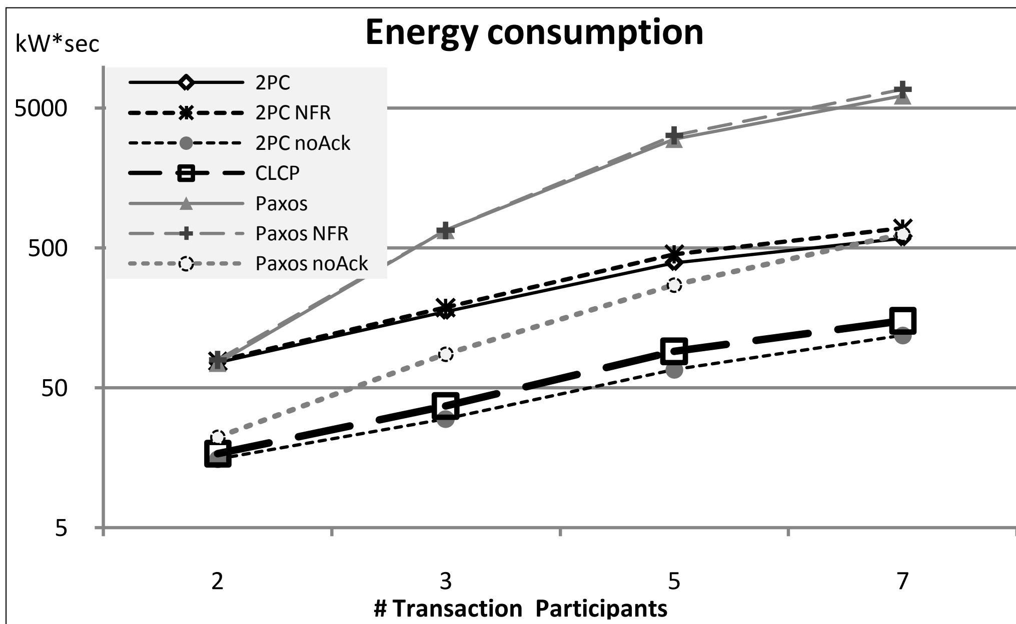

Figure 5.7: Attraction Point Mobility Model

The energy consumption for the NFR routing protocols does not differ significantly from Relay Routing using acknowledgement messages. Thus, NFR routing is not preferable compared to Relay Routing for the atomic commit protocol.

Although the mobility models differ in the total energy consumption required for the atomic commit protocols, we can see that the differences between the protocols used correlate for each mobility model. This is due to our transaction spawning model, since transactions are spawned among a set of participants, each participant of which is at most d field units away from at least one transaction participant. In each mobility model, the participants move during the atomic commit protocol execution such that they will likely separate.

Another important criterion is the time which each commit protocol takes to come to a commit decision. We measure the average blocking time of each participant, i.e. the time between sending the first voteCommit message and receiving the transaction decision. Figure 5.8, 5.9, and 5.10 show the results for each mobility model depending on the number of transaction participants.

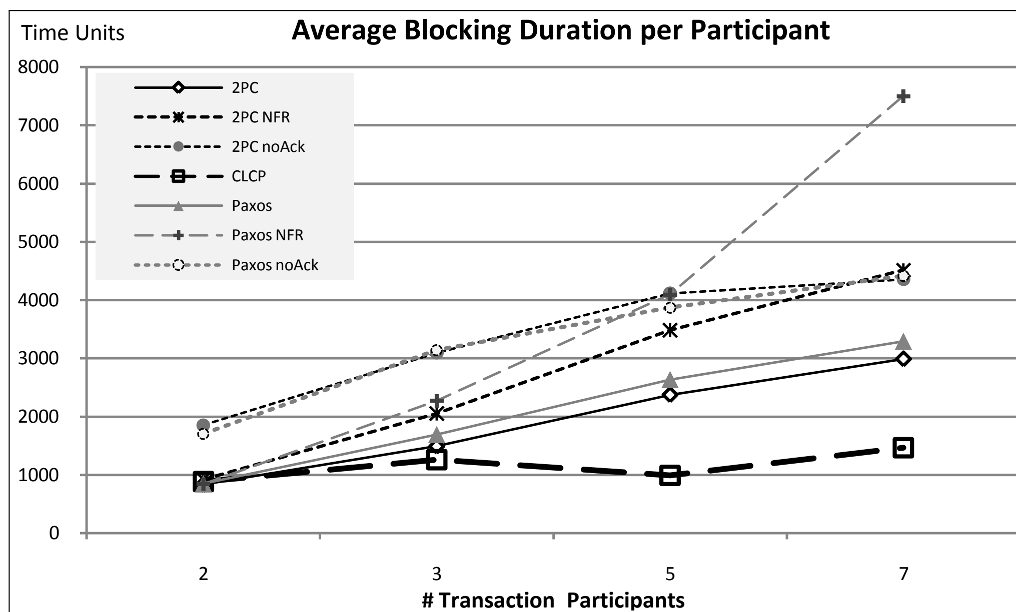

Figure 5.8: Manhattan Mobility Model

Again, the protocols omitting acknowledgements, i.e. “2PC noAck” and “Paxos noAck”, show the worst behavior. When using these protocols, the average blocking time per participant is significantly higher than when using the corresponding protocols having acknowledgement messages.

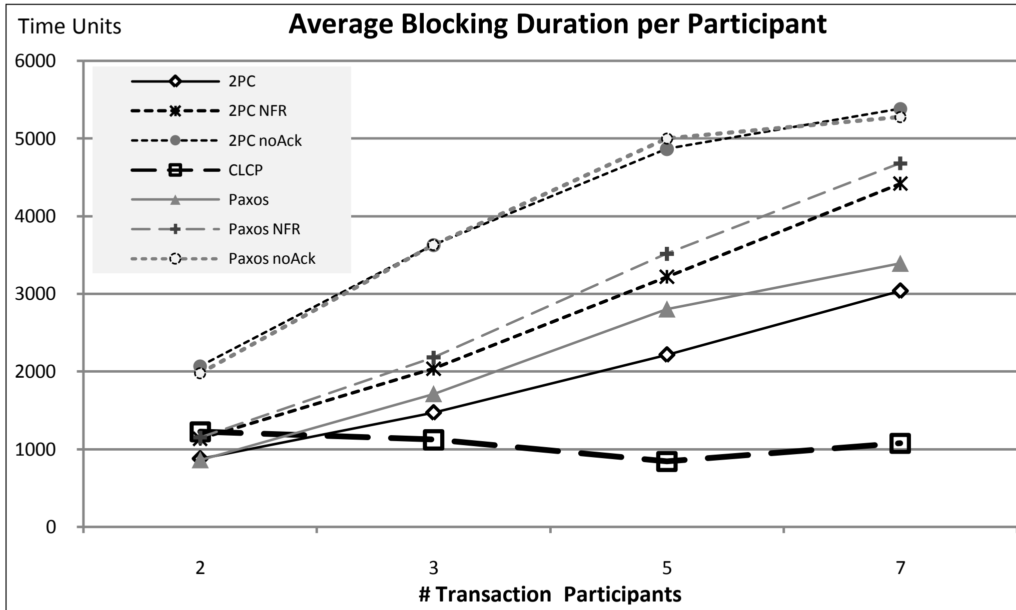

Figure 5.9: Random Waypoint Mobility Model

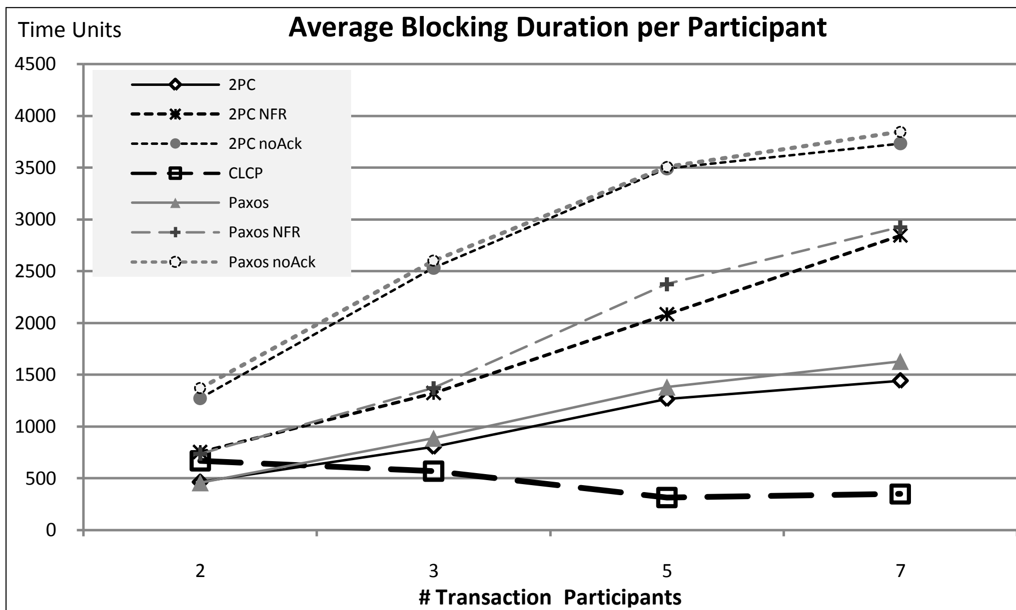

Figure 5.10: Attraction Point Mobility Model

Similarly, protocols using NFR routing show worse blocking times than those using Relay Routing and acknowledgements. Especially for the Manhattan Mobility Model, “Paxos NFR” shows the highest blocking times for 5 and more participants.

CLCP has the lowest blocking times for 3 and more participants, which is not only caused by the higher number of committed transactions, but also by the fast decentralized commit phase, which lets participants determine individually when a transaction has to be comitted, in contrast to the centralized protocols that rely on a coordinator.

5.5.9 Conclusion from the Experiments

Our experiments have confirmed that the use of acknowledgement messages is crucial for the overall number of committed transactions. However, when using ACKs, energy consumption grows. In contrast to 2PC and Paxos Commit, the energy consumption of CLCP is competetive with protocols that do not use ACKs, but CLCP allows to commit even more transactions than the 2PC and Paxos Commit protocols using ACKs.

When we compare Relay Routing with NFR, we see that Relay Routing shows a significantly higher number of committed transactions than NFR by using the same amount of energy. Thus, Relay Routing is generally preferable over NFR routing for atomic commit protocols.

Although the mobility model used has an influence regarding absolute energy consumption, the mobility model does not significantly influence the comparison between the atomic commit protocols.

Furthermore, we can see that CLCP has very short blocking times for 3 or more transaction participants, caused by both the number of committed transactions and the fast decentralized commit phase of CLCP.

5.6 Related Work

A lot of contributions aim for mobile environments that consist of both mobile devices and fixed infrastructure, some of them are [23, 38, 49, 56, 57, 72]. In contrast to all these approaches, our atomic commit protocol is designed with the assumption of a completely mobile ad-hoc environment without any fixed parts. As a consequence, we can neither rely on compensation transactions (they may never reach their destinations), nor shift work that is crucial for the protocol correctness

to fixed-wired, stable parts of the network. Thus, we consider our protocol being applied to completely mobile application scenarios.

Atomic commit protocol blocking that occurs during the atomic commit protocol execution is inevitable in asynchronous networks. This follows from contribution [63], which has proven that blocking during atomic commit protocol execution cannot be avoided if it cannot be determined whether a node has failed or is still working in another network partition. To enhance the Coordinator's availability of the atomic commit protocol, some contributions propose the use of protocols with more than one Coordinator. [58], for example, suggests the use of backup Coordinators in 2PC, [30] uses Paxos Consensus [40] to get a consensus on the commit decision, and [9, 12] allow "controlled failures" by proposing a combination of the different protocols 2PC, 3PC [62], and Paxos Consensus. The Chandra-Toueg Algorithm [17] also uses consensus to reach an agreement. Different from the version numbers used by Paxos Consensus, the Chandra-Toueg Algorithm uses rotating leaders that are determined by the number of the current round. However, [69] experimentally evaluated the Paxos Consensus and the Changra-Toueg Algorithm with the result that both algorithms have the same performance if neither crashes nor message loss occurs. The algorithms differ when failures must be handled. In such situations, Paxos Consensus has better performance since a participant takes over the leader role immediately after it has detected a leader failure, while the Chandra-Toueg Algorithm has a predefined order of leaders. The difference between both algorithms grows when failures occur simultaneously. Thus, CLCP uses a termination phase concept, whose version number approach is related to Paxos Consensus.

However, as our experiments have shown, consensus protocols operating on the application layer are expensive in terms of energy and need a special centralized main coordinator that coordinates the protocol.

Other contributions focusing on agents, e.g. [49], propose to shift the coordination workload to fixed parts of the network by using *participant-agents*, which are executed on base stations and responsible for sending the votes and accepting the commit decision. However, our extension is also usable for completely mobile networks without having a fixed infrastructure.

Another important problem of approaches operating on the ISO-OSI application layer (e.g. [43,61]) involves the guarantee of message delivery. In mobile networks, a single transmission of a message is not necessarily received by the next participant on the routing path, but this is assumed by the reception models of current atomic

commit protocols for mobile networks, e.g. [22, 38, 49, 72]. These protocols assume environment models using the unit-disc-model which implies that whenever a node sends a message, any node within its specified communication range receives the message, while any other node outside the disk does not. However, [26] has shown that a non-binary reception probability – the *quasi unit-disc model* – leads to very different results compared to the use of the unit-disc-model. In practice, additional acknowledgement messages are required in order to guarantee message delivery at least at the network layer. However, these additional acknowledgements consume energy.

In contrast, CLCP is able to use acknowledgements to piggyback additional information, which not only ensures correct message delivery but also causes the consensus algorithm to be competitive in terms of number of messages and energy used. Furthermore, CLCP does not rely on a special coordinator since it is a distributed protocol.

5.7 Summary and Conclusion

We have presented CLCP, a distributed cross layer commit protocol for mobile ad-hoc networks that uses consensus among all participants to come to a commit decision. We have shown how CLCP uses the Commit Matrix to decentralizedly come to a commit decision, and how the protocol terminates when network partitioning occurs. Furthermore, we have proven the correctness of CLCP, i.e. that all participants deciding on commit come to the same commit decision, and that CLCP terminates when a majority of participants is able to communicate with each other for a sufficiently long period of time.

Our experimental results show that CLCP, like 2PC without ACKs, consumes very little energy compared to the other protocols, but is superior to all the other protocols regarding the number of committed transactions. Therefore, we regard CLCP being a significant contribution to atomic commit protocols for mobile ad-hoc networks.

Chapter 6

Bi-State-Termination

6.1 Problem Description

In consequence of the protocol blocking definition 5.1.1 given in the last chapter, another serious problem arises, which has an effect on concurrent transactions:

Definition 6.1.1 A *transaction* T is *blocked* after a database proposed to execute T (e.g. by sending a voteCommit message) and waits for the final commit decision but is not allowed to abort or commit T unilaterally on its own.

Transaction blocking summarizes the unilateral impossibility to abort or commit a transaction, but does *not* mean that a transaction U waits to obtain locks from a concurrent transaction, since in this case U can be aborted by the database itself.

Example 6.1.2 *Assume 2PC is used and no problems occur. In this situation, transaction blocking occurs at all participating databases for a short period of time, namely within the time interval between the sending of the vote message and the receiving of the commit or abort message.*

While protocol blocking does not necessarily prevent the possibility of a unilateral abort by some of the databases, transaction blocking has an effect on concurrent transactions: They cannot be processed until the final commit decision is received. This can be explained as follows. A database that proposed to execute a transaction T_i has sent a result and a "vote for commit" message. Therefore, it must guarantee that the result is still valid when the coordination instance sends the commit instruction for T_i. However, since the coordinator may also send an abort message, the database cannot commit the transaction unless the transaction decision has been received. This, however, requires that any other concurrent transaction T_c that accesses in a conflicting way some or all tuples of T_i, cannot be executed and must wait until T_i is either committed or aborted. However, since the transaction is in a blocking situation, the database cannot terminate this situation on its own,

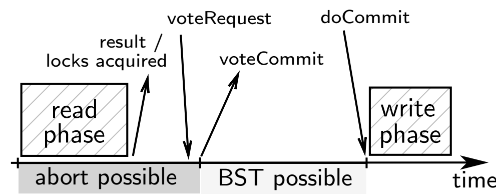

Figure 6.1: 2PC Transaction Execution

it must wait until it receives a commit/abort message for T_i. If this commit/abort message never reaches the database, T_i will never finish and thus T_c cannot commit. Furthermore, the blocking effect is independent of the used concurrency control mechanism since the database cannot let a transaction T_c that is in conflict with T_i commit without knowledge of the commit decision of T_i.

The problem of *infinite transaction blocking* for a transaction T_i occurs, if a database has sent a voteCommit message on T_i, but never receives the final commit decision, e.g. due to disconnection, movement, or network partitioning.

One might think that time-out based approaches solve this problem. However, if we add a time-limit for the commit decision, then the scenario fulfills the requirements of the coordinated attack scenario [28], in which the commit decision is that two generals use an unreliable communication channel to agree on a time for a common attack. For this scenario, [28] proves that a commit decision is not possible under the assumption that message loss may occur. The conclusion of this coordinated attack scenario is that the use of time-out votes (e.g. "my commit vote is valid until 3:23:34") does not allow the databases to unilaterally abort the transaction and therefore does not solve the problem of infinite transaction blocking.

6.2 Transaction Model Analysis

Definition 6.2.1 Assume a database in state S_0 executes a transaction T_i. Then, we call $\mathsf{Result}_{T_i}(S_0)$ the result value that the databases returns to the initiator after finishing the read phase of T_i.

Above the time line, Figure 6.1 shows the standard application of 2PC for a distributed transaction T_i when a locking mechanism like 2-Phase-Locking [24] is used. Below the time line, Figure 6.1 illustrates the possible database reactions in case the database does not receive expected messages after a timeout. As long as the database has not voted for commit, it can still abort T_i and release the locks.

Different from traditional 2PC, our protocol allows *Bi-State-Termination* (BST) to terminate a transaction even after the commit vote has been sent.

The transaction execution shown in Figure 6.1 involves two messages showing that the locked point was reached: Both the voteRequest message and the doCommit message indicate that all databases have obtained all necessary locks. However, for the purpose of ensuring serializability, it is only necessary to reach this locked point once, as protocol optimizations like "unsolicited vote optimization" [66] show.

As the transaction sequence is fixed after the voteRequest message has been received by each database, serializability is guaranteed. However, for ensuring strictness, i.e. to guarantee recoverability and to avoid cascading aborts, each database must hold all locks until the doCommit message is received and the write phase has been finished. Unfortunately, if the doCommit message is lost or cannot reach a database, the database must hold the locks and cannot abort the transaction on its own, which however, is possible before the vote for commit message has been sent.

In the remainder of this section, we develop a solution that not only unblocks and processes concurrent and depending transactions if the commit decision cannot be received by a database for a longer period of time. Our solution, which is called *Bi-State-Termination* (BST) of a transaction T_i, also guarantees atomicity. The main idea of BST is that if the coordinator's decision for a transaction T_i is delayed, a concurrent transaction T_c depending on T_i can be processed by transferring the required locks from T_i to T_c, and then by executing T_c on two database states: one state having T_i committed, the other state having T_i aborted. However, it is the database's choice whether or not and after which timeout it applies BST.

We want to achieve that all transactions belonging to a distributed global transaction are executed in an atomic fashion, and that each concurrent execution of different distributed global transactions is serializable, i.e. the execution produces the same output and has the same effect on the databases as some serial execution of the same distributed global transactions.

Definition 6.2.2 Assume that a database is in a state S_0 that has been created by some previous transactions before the write phase of T_i is executed. Assuming no concurrent transaction has changed the database state while T_i is executed, we call the database state that is caused by the write phase of T_i the state S_{T_i}. Let T_i consist of the sequence of operations $\mathrm{O}_{i_1}, \mathrm{O}_{i_2}, \ldots, \mathrm{O}_{i_n}$. When this sequence of operations is applied to the database, we call the changes that have been made by the operations the *delta of the transaction* T_i. We write $\Delta_{T_i}(S_0)$ if the sequence of operations $\mathrm{O}_{i_1}, \ldots, \mathrm{O}_{i_n}$ is applied on the database state S_0. When T_i has been

committed, the result of applying Δ_{T_i} to the database state S_0 becomes visible for other transactions, thus we get the new database state S_{T_i}, for which we write $S_{T_i} = S_0 \oplus \Delta_{T_i}(S_0)$.

Note, that when a transaction T_i, which, for example, increments integer values, is executed on two different database states S_0 and S_1, the transaction execution can lead to different deltas $\Delta_{T_i}(S_0)$ and $\Delta_{T_i}(S_1)$. However, when T_i does not include branches or loops, the sequence of operations remains the same for all executions.

Lemma 6.2.3 *Assume a database is in state S_0, a transaction T_i is executed, and a concurrent transaction T_j is started, but T_i does not depend on T_j and vice versa. Therefore, the changes of transaction T_j do not affect the execution of the transaction T_i. The equations*

$$\Delta_{T_j}(S_0) = \Delta_{T_j}(S_{T_i}) \quad \text{and} \quad \Delta_{T_i}(S_0) = \Delta_{T_i}(S_{T_j})$$

hold, which means that the modifications of a transaction T_j are independent of any previous modification of a non-dependent transaction T_i and vice versa.

Proof Assume the database state before the execution of T_i and T_j is S_0.

$$\begin{aligned}
& T_j \text{ does not depend on } T_i \text{ and} \\
& T_i \text{ does not depend on } T_j \\
\iff\quad & \forall \text{ tuple } t\, \forall \text{ attribute } a\, (\neg O_i \text{ accesses } t.a \vee \neg O_j \text{ accesses } t.a \\
& \vee (O_i \text{ reads } t.a \wedge O_j \text{ reads } t.a)) \quad \text{(follows from Def. 3.3.1 and 3.3.2)} \\
\implies\quad & S_0 \oplus \Delta_{T_i}(S_0) \oplus \Delta_{T_j}(S_{T_i}) \\
& = O_{j_n}(\ldots O_{j_2}(O_{j_1}(O_{i_n}(\ldots O_{i_1}(S_0))))) \\
& = O_{i_n}(\ldots O_{i_2}(O_{i_1}(O_{j_n}(\ldots O_{j_1}(S_0))))) \quad \text{(since operations do not conflict)} \\
& = S_0 \oplus \Delta_{T_j}(S_0) \oplus \Delta_{T_i}(S_{T_j}) \\
\implies\quad & \left(\Delta_{T_j}(S_0) = \Delta_{T_j}(S_{T_i})\right) \wedge (\Delta_{T_i}(S_0) = \Delta_{T}(S_{T_j}))
\end{aligned}$$

Therefore, whenever a set BT of transactions is blocked, the result of a transaction T may only be influenced by those transactions $DBT \subseteq BT$ on which T is dependent.

6.3 Solution

In the following, we focus on the question:

> What can a database D executing a transaction T_i do, when T_i is blocked, and a concurrent transaction T_c requests access to data tuples accessed by T_i in a conflicting way.

A proposed solution to answer this question can be found in standard literature for databases, and is quite simple: wait. However, during the wait, the number of transactions that wait concurrently may increase if the blocking continues. Another possibility is to abort the concurrent transaction T_c. Although correct, this behavior is not satisfying.

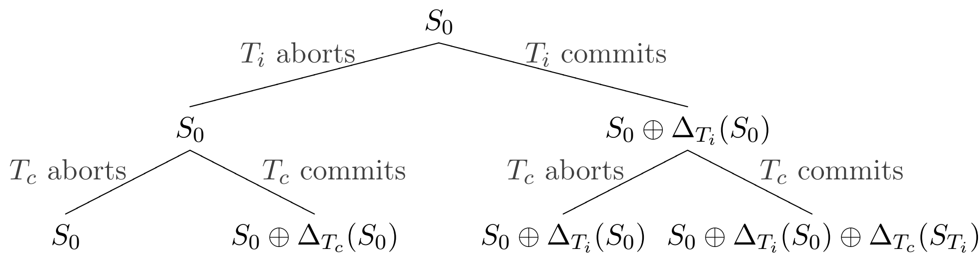

Figure 6.2: Possible database states if T_i and T_c conflict and block

Our solution called *Bi-State-Termination* is based on the following observation: Whenever transaction blocking occurs, the database does not know whether a transaction T_i waiting for the commit decision will be aborted or committed. However, only if the transaction is committed, the database state changes. Let S_0 denote the database state before T_i was executed. Although the database does not know the commit decision for T_i, it knows for sure that either S_0 or $S_0 \oplus \Delta_{T_i}(S_0)$ is the correct database state, depending on the commitment of T_i. Figure 6.2 shows these two possible states in the tree. With this knowledge, the database can try to execute a concurrent conflicting transaction T_c on both states S_0 and $S_0 \oplus \Delta_{T_i}(S_0)$. Whenever the two executions of T_c on S_0 and on $S_0 \oplus \Delta_{T_i}(S_0)$ return the same results to the Initiator, i.e. $\mathsf{Result}_{T_c}(S_0) = \mathsf{Result}_{T_c}(S_{T_i})$, T_c can be committed regardless of T_i, even though they are conflicting. Otherwise it is the application's choice whether it handles two possible transaction results. However, since T_c depends on T_i, we might have $\Delta_{T_c}(S_0) \neq \Delta_{T_c}(S_{T_i})$. Therefore, the database must store both deltas $\Delta_{T_c}(S_0)$ and $\Delta_{T_c}(S_{T_i})$, and if T_c commits before T_i is committed or aborted, the database knows that either the state $S_0 \oplus \Delta_{T_c}(S_0)$ is valid, or the state $S_0 \oplus \Delta_{T_i}(S_0) \oplus \Delta_{T_c}(S_{T_i})$ is valid. Figure 6.2 shows the execution tree with both T_i and T_c being blocked. The leaves represent the database states that may be valid depending on the decisions for the blocked transactions T_i and T_c.

6.3.1 Bi-State-Termination

Let $\Sigma = \{S_0, \ldots, S_k\}$ be the set of all legal possible database states for a database D. A *traditional transaction* T_i is a function $T_i : \Sigma \mapsto \Sigma$, $S_a \to S_b$, which means the resulting state S_b of T_i depends only on the state S_a on which T_i is executed.

A *Bi-State-Terminated transaction* T_i is a function $BST : 2^{\Sigma} \mapsto 2^{\Sigma}$,

$$\underbrace{\{S_i, \ldots, S_j\}}_{\text{Initial States}} \to \underbrace{\{S_i, \ldots, S_j\}}_{T_i \text{ aborts}} \cup \underbrace{\{T_i(S_i), \ldots, T_i(S_j)\}}_{T_i \text{ commits}}$$

that maps a set $\Sigma_{\text{Initial}} \subseteq \Sigma$ of Initial States to a super set $\Sigma_{\text{Initial}} \cup \{T_i(S_x) | S_x \in \Sigma_{\text{Initial}}\}$ of new states, where $T_i(S_x)$ is the state that is reached when T_i is applied to S_x.

This concept of Bi-State-Termination leads to the following commit decision rules for the transaction execution of a transaction T_i on a database *DB*:

DB checks T_i's dependency on concurrent blocking transactions. The following situations may occur: T_i is independent of all currently blocked transactions. Then, T_i can be executed immediately.

Otherwise, T_i depends on a set $\{T_j \ldots T_n\}$ of blocked transactions. As T_i depends on each of the transactions $\{T_j \ldots T_n\}$, each of them has reached its lock point before T_i. Thus, for any concurrent execution of $\{T_j \ldots T_n\}$, there is an equivalent serial execution *ESE* of $\{T_j \ldots T_n\}$. As *ESE* only has to reflect the order in which transactions leave the lock point, *ESE* can always be constructed, as described below. Thus, serializability is guaranteed for $\{T_j \ldots T_n\}$. Then, *DB* can *Bi-State-Terminate* the transactions $\{T_j \ldots T_n\}$, and executes the transaction T_i on all possible combinations of abort and commit decisions of the transactions $\{T_j \ldots T_n\}$ in *ESE*.

The serializable sequence *ESE* of the transactions $\{T_j \ldots T_n\}$, on which T_i is executed, must obey the following conditions. For each pair (T_j, T_n) of the transactions for which a dependency $T_j \to T_n$ exists, T_j left its lock point before T_n left its lock point. However, in order to execute the transaction T_n that depends on the blocked transaction T_j, the transaction T_j must have been Bi-State-Terminated. In this case, the order $(T_j < T_n)$ is fixed.

Note that if there is no dependency $T_j \to T_n$, and no dependency $T_n \to T_j$, the execution sequence of the blocked transactions (T_j, T_n) does not matter since the transactions are independent of each other, cf. Lemma 6.2.3.

This means, the transaction T_i must only be executed on all combinations of commit and abort decisions for the blocked transactions $T_j \ldots T_n$, but not on all possible sequences (permutations) of the transactions $T_j \ldots T_n$.

If T_i must be executed on multiple database states, this might yield different results. Let $S_1 \ldots S_{2^n}$ be the states that can be reached by any combination of commit/abort decisions on the n transactions $\{T_i \ldots T_n\}$ that are Bi-State-Terminated. If $\mathsf{Result}_{T_i}(S_0) = \ldots = \mathsf{Result}_{T_i}(S_{2^n})$ holds, transaction T_i can be committed since it has a unique result. Otherwise, the application that initiated T_i can choose whether

- it aborts T_i completely,
- it commits T_i and deals with multiple possible results,
- it aborts or commits only some transaction execution *branches* that are based on certain depending transactions. For example, the application may specify that T_i should only commit when T_k aborts, and that T_i should abort otherwise,
- it waits.

When T_i or a single execution branch of T_i commits, the database merges the corresponding delta of T_i with the possible branch state.

Example 6.3.1 *Assume T_c depends on T_i, but, different from Figure 6.2, T_c should only commit when T_i commits and otherwise abort. As illustrated in Figure 6.3, this would only affect the leftmost and rightmost branches of Figure 6.2. Therefore, $\Delta_{T_i}(S_0)$ in Level 1 is replaced with $(\Delta_{T_i}(S_0) \oplus \Delta_{T_c}(S_{T_i}))$ since, in this case, a commit of T_i automatically means a commit of T_c. Note that in this example, the commit* decision *for T_c is made* before *the decision of T_i, but the* execution *sequence is the other way round, namely $T_i < T_c$. Furthermore, the tree is flattened one level since only T_i is yet blocked.*

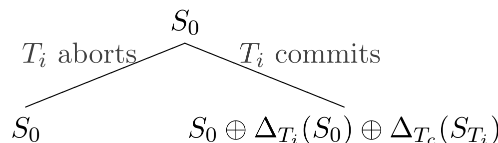

Figure 6.3: T_c should commit only if T_i commits

6.3.2 Complexity

It can be seen that the complexity of the Bi-State-Termination of T_i depends on the number of blocked transactions b, and that BST has a complexity of $O(2^b)$

database states. However, our implemented solution uses a compact data structure and optimizes read and write operations in such a way that each transaction operation must only be executed once, regardless of the number of blocked transactions. Although, in the worst case, the number of tuples may grow exponentially, standard database query optimization techniques can be fully applied.

6.3.3 Correctness

Theorem 6.3.2 *Bi-State-Termination in combination with 2-Phase-Locking guarantees serializability.*

Proof As our solution uses Two-Phase Locking (2PL) and 2PL is proven to guarantee serializability according to [6], we show that Bi-State-Termination does not change the order of pairs of conflicting operations of transactions given by 2PL: Our transaction execution involves one point, namely the lock point, where each transaction that belongs to a global transaction must hold all locks. This means the request to vote on a transaction's commit status can only be sent by the coordinator when all databases acquired the necessary transaction locks, which the databases indicate by sending the transaction result. The sequence of transactions is fixed at that time when each transaction enters its lock point. Although Bi-State-Termination may release locks after this lock point, the release of locks does not change the order of transactions for the following reason. A transaction T_c that gets locks from a Bi-State-Terminated transaction T_i is either executed after T_i has been committed ($T_i < T_c$) or T_i is aborted.

Note that although the commit command for T_c may be issued before the commit command of T_i, the order of applying the transactions on the database is still $T_i < T_c$.

6.4 BST Rewrite Rules

Our BST rewrite rule system modifies each database relation in such a way that it gets an extra column *"Conditions"* that describes for each database tuple the condition under which it is regarded as being true. Furthermore, the database contains a single table *"Rules"* storing rules that relate these conditions to each other.

Whenever currently active transactions insert, delete or update tuples in a relation R, whether or not a tuple t will finally belong to R depends on the commit or

abort decision of these transactions. We use a condition in order to express which of the active transactions must commit and which must abort, such that a tuple t finally belongs to a relation R. In our implementation, each relation R is augmented by an extra column "Conditions" that, for each tuple t, stores the condition under which it finally belongs to R.

This can be implemented by the following rewrite rule that modifies the create table command for database relations:

```
create table R ( <column definitions> )
  ⇒ create table R'( <column definitions, string conditions>)
```

6.4.1 Status Without Active Transactions

When all transactions are completed either by commit or by abort, the column "Conditions" contains the truth value "true" for each tuple in each relation of the database.

6.4.2 Write Operations on the BST Model

Insertion

Whenever a tuple $t = (\text{value}_1, \ldots, \text{value}_N)$ is inserted into a relation R by a transaction with transaction identifier T_i, we implement this by inserting $t' = (\text{value}_1, \ldots, \text{value}_N, T_i)$ into the relation R', i.e. the database system implementation applies a rewrite rule:

```
insert into R values (value_1,...,value_N)
⇒ insert into R' values (value_1,...,value_N, T_i ).
```

The idea behind the condition T_i is to show that the tuple t belongs to the database relation R if and only if transaction T_i will be committed.

Deletion

Whenever a tuple $t = (\text{value}_1, \ldots, \text{value}_N)$ is deleted from a relation R by a transaction with transaction identifier T_i, we look up the tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C)$ representing the tuple t, where C is the condition under which t belongs to the database relation R.

We implement the deletion of t from R by the transaction T_i by replacing the condition C found in t' with a condition C_2 and by adding a logical rule to the

table rules stating that C_2 is true if and only if C is true and T_i is aborted. For this purpose, the database system applies the following rewrite rule, where $A_1, \ldots, A_N$ denote the attributes of R for the values (value$_1, \ldots,$ value$_N$):

```
delete t from R where t.A1=value1,...,t.AN=valueN
⇒ update t' in R' where t.A1=value1,...,t.AN=valueN set condition=C2;
insert into rules values ( C2 , C1 and not Ti )
```

The idea behind this rewriting is the following. (not T_i) represents the condition that transaction T_i will be aborted. The inserted rule states that C_2 is true if C_1 is true and T_i will be aborted. After the update operation, we have a tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C_2)$ in R' which represents that fact that t belongs to R if and only if C_2 is true, i.e. if C is true and T_i is aborted.

Update

An update of a single tuple is simply executed as a delete operation followed by an insert operation.

Set-Oriented Write Operations

When a transaction inserts, updates, or deletes multiple tuples within a single operation, this can be implemented by a collection of individual insert, update, or delete operations.

Completion of a Transaction

When transaction T_i is completed with commit, the condition T_i is replaced with true in each rule in the rules table and in each value found in the column "Conditions" of a relation R'. However, when T_i is completed with abort, T_i is replaced with *false* in each rule found in the rules table, and each tuple of R' containing the value T_i in the column "Conditions" is deleted.

Furthermore, rules that contain the truth value *true* or *false* are simplified. Whenever this results in a rule (C, *true*) or in a rule (C, *false*), then C itself is replaced with the value "true" or "false" respectively. Other rules that contain C are simplified as well. Furthermore, all tuples t' in which C occurs are treated as follows. If the rule is (C, *true*), the value C is replaced with true in each tuple t' in which C occurs in the column "Conditions". However, if the rule is (C, *false*), each

tuple t' in which C occurs in the column "Conditions" is deleted. Finally, rules (C, *true*) or (C, *false*) are deleted from the relation "Rules".

Example

Consider the following transactions:

T_1: insert "Miller"
T_2: delete "Mitch"
T_3: change "M" to "R"

Line	ID	Name	Condition	Comment
1	(1)	Mitch		Initial
2	(1)	Mitch		Content after BST of T_1
3	(2)	Miller	C_1	
4	(1)	Mitch	C_2	Content after BST of T_1, T_2
5	(2)	Miller	C_1	
6	(1)	Mitch	C_3	Content after BST of T_1, T_2, T_3
7	(2)	Miller	C_4	
8	(3)	Ritch	C_5	
9	(4)	Riller	C_6	

Table 6.1: Content after Bi-State-Terminating T_1, T_2, and T_3

Line 1 of Table 6.1 represents the Initial database, lines 2-3 show the whole table content after BST of T_1, lines 4-5 represent the table content after BST of T_1 and T_2, while lines 6-9 show the table after BST of T_1, T_2, and T_3. The conditions are linked to the Rules Table 6.2, which shows the concrete conditions for which each data tuple referenced by Condition C_j is valid. The condition C_4, for example, is fulfilled when T_1 commits and T_3 aborts. In this case, line 7 of Table 6.1 becomes valid.

6.4.3 Read Operations on the BST Model

Whenever a read operation on R is implemented by a read operation on R', the conditions are kept as part of the result. The relational algebra operations are implemented as follows.

ID	Condition	Comment
–	–	Initial
C_1	T_1	Content after BST of T_1
C_1	T_1	Content after BST of T_1, T_2
C_2	$\overline{T_2}$	
C_3	$\overline{T_2 T_3}$	Content after BST of T_1, T_2, T_3
C_4	$T_1\overline{T_3}$	
C_5	$\overline{T_2}T_3$	
C_6	T_1T_3	

Table 6.2: Rules Table after Bi-State-Terminating T_1, T_2, and T_3

Selection

Each selection with selection condition SC that a query applies to a relation R, will be applied to R', i.e. the database system applies the following rewrite rule to each selection:

SC(R) ⇒ SC(R')

Duplicate Elimination

Duplicate elimination is an operation that is used to implement projection and union. When duplicates occur, their conditions are combined with the logical OR operator. That is, given the relation R' contains two tuples $t'_1 = (\text{value}_1, \ldots, \text{value}_N,$ $C_1)$ and $t'_2 = (\text{value}_1, \ldots, \text{value}_N, C_2)$ these two tuples are deleted and a single tuple $t' = (\text{value}_1, \ldots, \text{value}_N, C_{C12})$ is inserted into R, and a rule $(C_{C12}, C_1$ or $C_2)$ is inserted into the rules relation.

Set Union

Set union of two relations R_1 and R_2 is implemented by applying duplicate elimination to the set union of R'_1 and R'_2. The database system applies the following rewrite rule:

$R_1 \cup R_2 \Rightarrow$ removeDuplicates($R_1' \cup R_2'$)

Projection

Projection of a relation R_1 on its attributes $A_1, \ldots, A_N$ is implemented by applying duplicate elimination to the result of applying the projection to R_1' including the column "Conditions". The database system applies the following rewrite rule:

P(A_1,...,A_n) (R_1) ⇒ removeDuplicates(P(A_1,...,A_n, conditions) (R_1'))

Cartesian Product

Whenever the cartesian product $R_1 \times R_2$ of two relations R_1 and R_2 must be computed, this is implemented using R_1' and R_2' as follows. For each pair (t_1', t_2') of tuples $t_1' = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R_1' and $t_1' = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$ of R_2', a tuple $t_{12}' = (\text{value}_1, \ldots, \text{value}_N, \text{value2}_1, \ldots, \text{value2}_N, C_{C12})$ is constructed and stored in $(R_1 \times R_2)'$. The database system applies the following rewrite rule:

$R_1 \times R_2 \Rightarrow (R_1 \times R_2)$' where ($R_1 \times R_2$)' can be derived by computing the set { (t_1,t_2,C_{C12}) | (t_1,C_1) ∈ R_1' and (t_2,C_2) ∈ R_2' } and by adding a rule (C_{C12}, C_1 and C_2) for each pair of C_1 and C_2 to the rules relation.

Set Difference

Whenever the set difference $R_1 - R_2$ of two relations R_1 and R_2 must be computed, this is implemented using R_1' and R_2' as follows. The set difference contains all tuples $t_1' = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R_1' for which no tuple $t_2' = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$ of R_2' exists, and furthermore, it contains a tuple $t_{12}' = (\text{value}_1, \ldots, \text{value}_N, C_{C12})$ for each tuple $t_1' = (\text{value}_1, \ldots, \text{value}_N, C_1)$ of R_1' for which a tuple $t_2' = (\text{value2}_1, \ldots, \text{value2}_N, C_2)$, $C_2 \neq C_1$, of R_2' exists. The condition C_{C12} is *true* if and only if (C_1 and not C_2) is *true*. The database system applies the following rewrite rule:

R_1 - R_2 ⇒ R_1' - R_2' where (R_1' - R_2') can be derived by computing the union of the following sets S_1 and S_2: S_1 = { (t_1,C_1) | (t_1,C_1) ∈ R_1' and not exists C_2 such that (t_1,C_2)∈ R_2' } S_2 = { (t_1,C_{C34}) | (t_1,C_3)∈ R_1' and exists C_4 such that (t_1, C_4)∈ R_2' and $C_3 \neq C_4$} and by adding a rule (C_{C34}, C_3 and not C_4) for each pair of C_3 and C_4 used in S_2 to the rules relation.

Other Algebra Operations

Other operations of the relational algebra like join, intersection, etc. can be constructed by combining the implementation of the basic operations. Of course, query optimization of operations like join etc. is possible.

6.5 Implementation

We have implemented BST in three versions and have compared their performance using a stress test. The first implementation, called *BST-Disk*, uses the rule table and rewrite rules as stated in Section 6.4, and stores the rule table as a separate database table on disk.

A modification of this concept, the *BST-RAM* implementation, stores the rule table completely within main memory. This makes the rule table management faster but also susceptible to failures like power failure.

The third implementation, called *Fast-BST*, does not use a separate rule table anymore. Instead, Fast-BST adds an additional column "Condition" of type string to each table, which stores the conditions under which the corresponding data row becomes valid. Thus, conditions need not be derived from the rules table; each tuple contains its conditions within the "Condition" column. Thus, Fast-BST makes rule table lookups to derive the conditions under which a tuple becomes valid superfluous and Fast-BST speeds up write operations that operate on many tuples for the following reason: The database does not need to generate and associate unique IDs to replaced conditions, it can update the "Condition" column in one pass by concatenating its value with the transaction ID. Furthermore, Fast-BST is not susceptible to power failures as BST-RAM.

In the following, we describe the Fast-BST implementation that is used in the evaluation.

6.5.1 Fast-BST – Write Operations

Fast-BST implements the concept of BST as follows. Fast-BST stores the before image and the after image of tuples that have been modified by BST transactions. For this purpose, the Fast-BST adds the column "Conditions" to each table that it uses, cf. Table 6.3.

The following step is executed for each insert statement `INSERT <data> INTO ...` of a transaction T_i:

	ID	Attributes	Conditions
(1)	1	α_1	
(2)	2	α_2	

Table 6.3: Original table containing the additional column "Conditions"

Algorithm 11 Implementing Inserts

1. Insert `<data>` into the corresponding table, and add T_i to the column "Conditions" of the newly inserted data

---

For each delete statement `DELETE ... WHERE <X>` of a transaction T_i, the following rewriting is necessary:

Algorithm 12 Implementing Deletes

1. Add the substring $\overline{T_i}$ to each entry in the column "Conditions" of each row where `<X>` evaluates to true. Simplify, if T_i wants to delete a tuple that T_i has just inserted before.

Algorithm 13 describes the update operation for the update statement `UPDATE ... WHERE <X>`.

Algorithm 13 Implementing Updates

1. Copy the tuples for which `<X>` evaluates to true into new data tuples, and concatenate T_i to the existing entries of the "Conditions" attribute of the new tuples. Update the newly copied tuples according to the update statement.

2. Add $\overline{T_i}$ to each entry in the column "Conditions" of each row where the corresponding entry in "Conditions" does not contain T_i and `<X>` evaluates to true.

Example 6.5.1 *Assume we execute the following sequence of three transactions T_1, T_2, and T_3, each containing one update statement, on Table 6.3.*

T_1: *`UPDATE Table1 SET Attributes=`α_3 `WHERE ID=1`*

T_2: *`UPDATE Table1 SET Attributes=`α_4 `WHERE ID=2`*

T_3: *`UPDATE Table1 SET Attributes=`α_2*
`WHERE (Attributes=`$\alpha_3 \vee$ `Attributes=`α_4`)`

Table 6.4 shows the result when all of the distributed transactions $T_1 \dots T_3$ block and Bi-State-Terminate.

	ID	Attributes	Conditions
(1)	1	α_1	$\overline{T_1}$
(2)	2	α_2	$\overline{T_2}$
(3)	1	α_3	T_1, $\overline{T_3}$
(4)	2	α_4	T_2, $\overline{T_3}$
(5)	1	α_2	T_1, T_3
(6)	2	α_2	T_2, T_3

Table 6.4: Content of Table 1 after Bi-State-Terminating T_1, T_2, and T_3

Fast-BST marks all tuples changed by transaction T_1 as before image (line (1)), copies them to line (3), and executes the update (on line (3)). Note that Table 6.4 shows the result when all transactions block, therefore it already contains the entry for T_3 in line (3). The same algorithm is applied for T_2. When T_3 is executed and both transactions T_1 and T_2 block, T_3 depends on T_1 and T_2. However, FAST-BST does not explicitly check for this dependency. In our example, T_3 only modifies data when either T_1 or T_2 commit. This dependency is maintained automatically by Step 1 of Algorithm 13 since the `<condition>` *of the update statement of T_3 is only* true *in lines (3) and (4). Then, these two rows are copied to the rows in line (5) and line (6), and T_3 is added to the "Conditions" column of each of these rows.*

6.5.2 Fast-BST – Read-Operations

Read operations are modified in the following way: Each value of the returned result additionally contains the corresponding values of the "Conditions" column. Then, the read operation must be processed by the database only a single time, regardless of the number of depending blocked transactions. However, the result R is not directly returned to the application, Fast-BST first checks whether R contains any entries in the "Conditions" column. If this is the case, it is the application's choice whether it handles these multiple uncertain results, or whether the application delays the read operations until the transactions listed in the "Conditions" column of R have been committed. If the application can handle multiple results, we can reduce the amount of transferred data by returning an object that creates the different possible valid database states directly within the application by means of the "Conditions" column.

6.5.3 Commit and Abort

The following rules apply when a blocked transaction T_i commits or aborts:

T_i **commits:** Delete all rows that contain $\overline{T_i}$ in the column "Conditions". Delete the string T_i from all entries within the "Conditions" column of the table.

T_i **aborts:** Is treated as $\overline{T_i}$ commits.

Example 6.5.2 *Assume T_3 commits. In this case, lines (3) and (4) are deleted from Table 6.4. Furthermore, the string T_3 must be deleted in Lines (5) and (6) from the attribute values for the column "Conditions", since a commit of T_1 or T_2 automatically implies that the changes of T_3 become valid.*

Note that the data set increases only temporarily and collapses to the original size when the commit decision for the Bi-State-Terminated transactions is known. For example, when the database receives the commit decisions for T_1 and T_3, the database knows the exact unique value for the data tuple with ID 1, which corresponds to Line (5) in case T_3 and T_1 commit, and to Line (1) in case T_3 commits and T_1 aborts.

6.6 Experimental Evaluation

We choose the TPC-C benchmark [36] for generating the test data and transactions. The following questions motivate our experimental evaluation: Which BST implementation is faster? How many transactions can be Bi-State-Terminated until the execution time and database size for following transactions is unacceptable? How does BST affect the overall transaction throughput and transaction execution time, when a certain percentage of transactions block?

6.6.1 BST Stress Test

To compare the three BST implementations and to determine how many blocked transactions per tuple each BST implementation can handle, we have executed a stress test. For this stress test, we have generated a database table consisting of a single data tuple. We have sequentially executed a number of database transactions on this table that do not finish, i.e. the transactions do not get a commit decision and thus block. Each of these blocked transactions has incremented or decremented the same data that is initially present. In order to be able to process further transactions, we have used our three BST implementations to terminate each blocked transaction. For this reason, each blocked transaction has doubled the number of

possible database states, and thus the number of possible values for the initial tuple grows exponentially for the number of blocked transactions. We have measured the time for processing the $(n+1)^{\text{th}}$ transaction when n transactions are blocked and terminated by BST for each BST implementation.

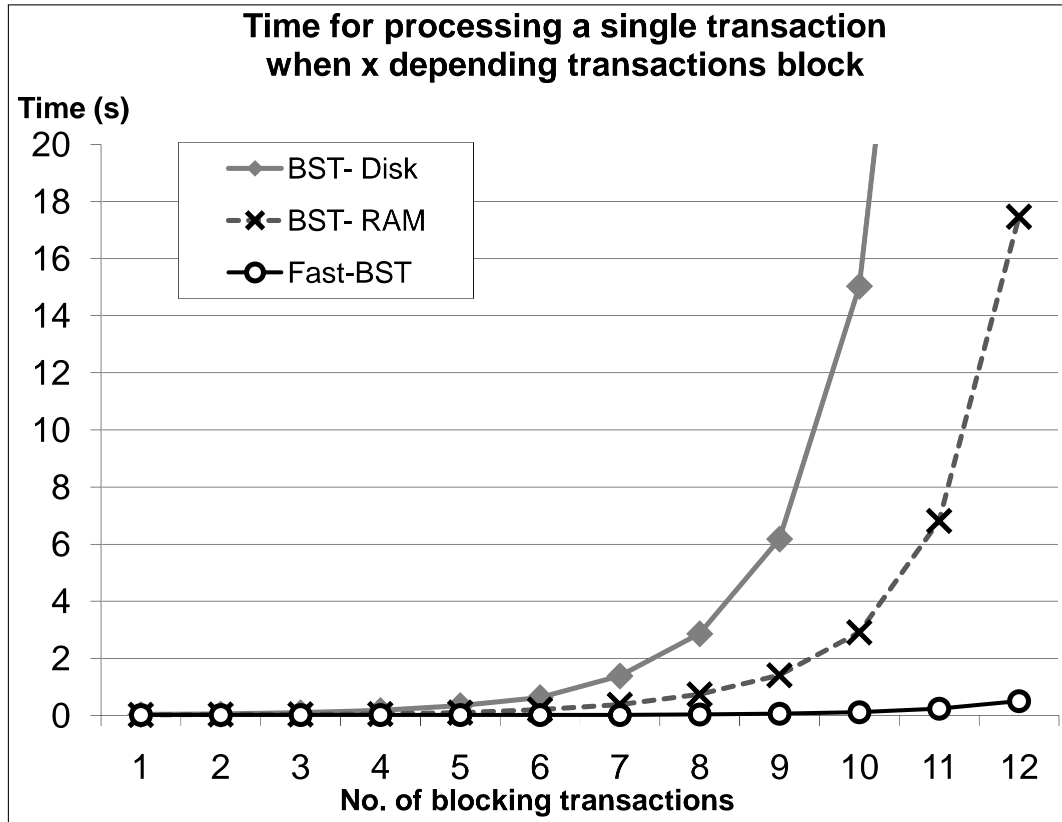

Figure 6.4: BST Stress Test – Performance

The y-axis of Figure 6.4 indicates the required time to process a single update transaction when the number of transactions indicated on the x-axis is terminated by BST. As all of these blocked transactions depend on each other, and all of them are coded to modify the initial tuple, the resulting growth in time and space is exponential. However, as the test indicates, the processing of a transaction when 10 blocked transactions have been terminated by BST does not take a large overhead for the Fast-BST implementation.

Both implementations that use a separate "Rules" table, i.e. BST-Disk and BST-RAM, are significantly slower than the Fast-BST implementation. The reason is that when transactions update a lot of data tuples, the corresponding condition must be derived from the rule table for each updated data tuple, and a new condition

ID must be generated and assigned separately to each data tuple. In comparison, Fast-BST only adds the transaction's ID to the "Conditions" column of all updated tuples, which can be done much faster.

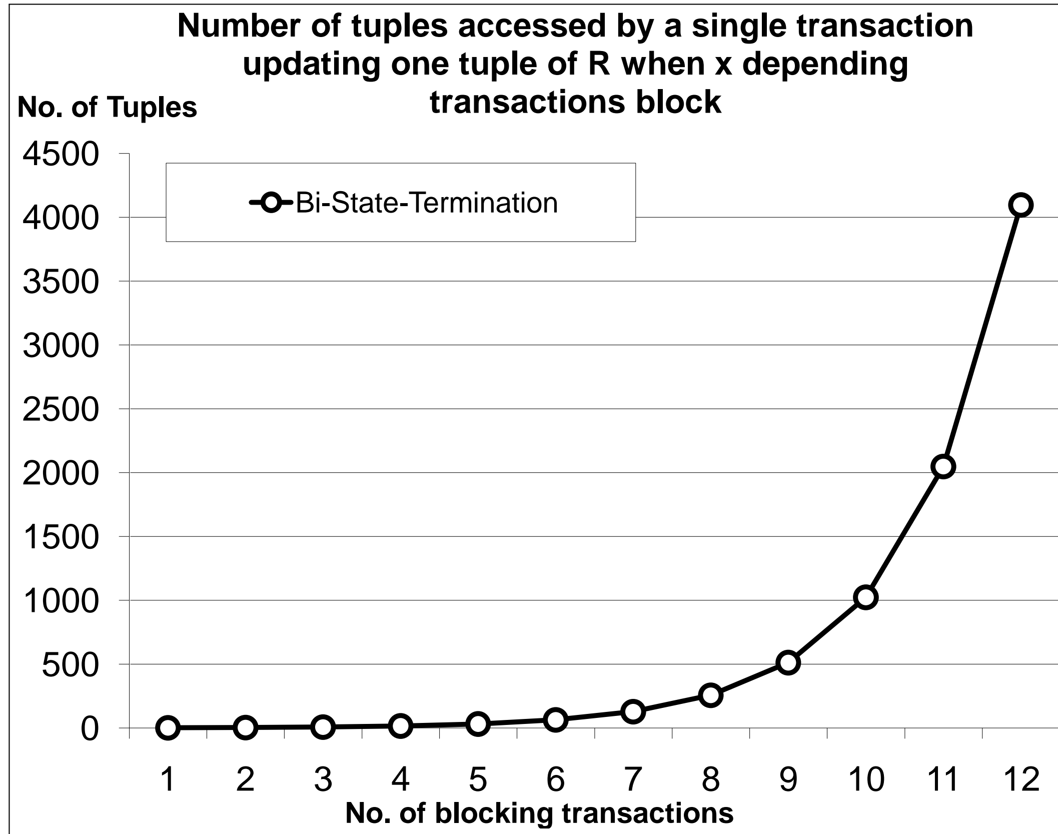

Figure 6.5: BST Stress Test – Space

On the y-axis, Figure 6.5 shows the number of tuples that are generated by BST for a single update operation, while the number of transactions indicated on the x-axis has been terminated by BST. Note that our BST implementations do not differ in the number of resulting tuples. Although the database's size grows exponentially each time a blocked transaction is terminated by BST, it is the database's decision to use BST for a transaction T_i or to wait until the decision for transactions on which T_i depends is known.

As we have seen, using the Fast-BST implementation allows the database to terminate more blocked transactions without a significant loss of performance. For this reason, we use the Fast-BST implementation in the following TPC-C benchmark test.

6.6.2 BST TPC-C Test

The TPC-C benchmark [36], an online transaction processing benchmark test, simulates an online-shop-like environment in which users execute order transactions against a database. The transactions additionally include recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

We used a TPC-C "scaling factor" of 2, which results in 139 MB of data and a total amount of 294 transactions, 41,8% of them being update operations. Characteristic for our implementation of the TPC-C benchmark is that the involved update transactions operate on a set of data tuples whose cardinality is low (i.e. 2 tuples), so we can expect a lot of conflicting write transactions. In order to simulate transaction blocking, a separate coordinator instance coordinates each transaction and delays the commit command based on different parameters in order to simulate blocked distributed transactions. For example, to simulate a transaction blocking of 1% of all transactions, we delayed the commit command of each 100th transaction.

Figure 6.6 shows the sum of all successfully committed transactions on the y-axis. On the x-axis, the overall time is shown. The different curves indicate whether BST was enabled, and they vary in the percentage of blocked transactions. Note that due to our simulated hotspot, a huge amount of transactions depend on each other. We can see that BST-enabled transaction processing is able to commit a lot more transactions than BST disabled transaction processing.

Note that the additional space used by BST is rather low, i.e., in our TPC-C experiments, BST requires only about 2% more space.

6.6.3 Evaluation Summary

We have run a stress test to compare three implementations for BST. While the BST-Disk and BST-RAM implementation use a separate rule table in order to manage the dependencies of the before- and after-images of the transactions, the Fast-BST implementation directly annotates the rule under which each data tuple becomes valid to the data row. The Fast-BST implementation is able to cope with 10 blocked transactions that all write on the same tuple without causing a loss of performance, while the BST-Disk and BST-RAM implementations can only handle 5 to 7 blocked transactions within reasonable time.

When setting up a transaction scenario, two extreme scenarios are possible that influence the outcome of BST: In the first scenario, each transaction operates on different tuples that are not accessed by any other transaction. In the second scenario, each transaction operates on the same tuples. As transaction blocking in

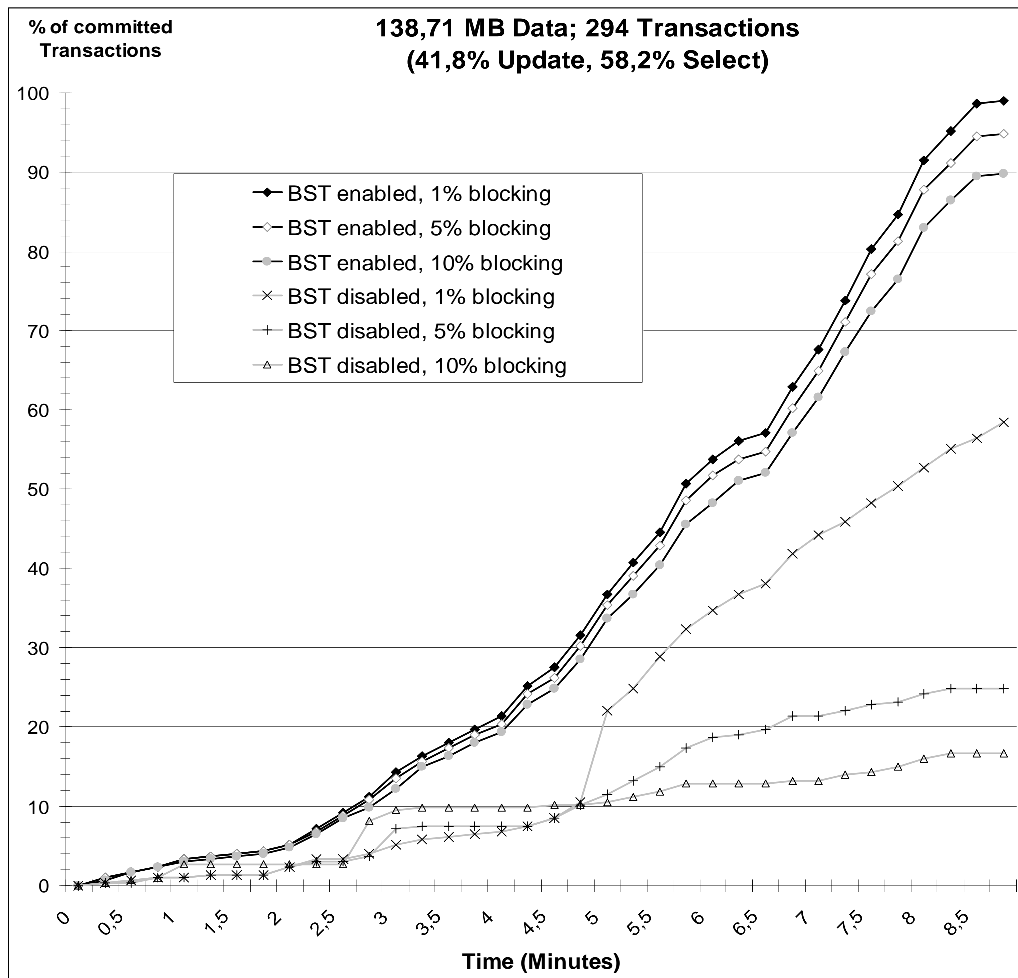

Figure 6.6: BST evaluation on the TPC-C benchmark

the first scenario does not have any influence on other transactions, enabling BST does not commit more transactions than disabling BST. In the second scenario, a blocked transaction would immediately prevent all following transactions from being processed. In this scenario, BST would allow the commitment of almost all transactions, while disabling BST would result in a total blocking situation. Due to these two possible scenarios, we used the TPC-C benchmark, which simulates a typical environment, for further experiments. We have preferred the Fast-BST implementation, which is able to enhance the amount of committed transactions in our TPC-C benchmark by 40 to 70%, depending on the number of blocked transactions.

Finally, note that if no more space is available or the required processing time grows, the database can decide for each individual transaction whether to use BST or to wait for the commit decision as in 2PC. In other words, our solution does not force the database to accept long execution times, and BST-enabled transaction processing never blocks more transactions than traditional transaction processing.

6.7 Related Work

Our solution relates to three ideas that are used in different contexts: Escrow locks [31], speculative locking [59], and multiversion databases [16, 18, 34].

Escrow locks are a refinement of field calls, which are used in environments where data hotspots are frequently accessed. The escrow lock calculates an interval $[i, k]$ for an attribute a by means of the currently processed updates. The interval indicates the actual upper and lower boundary that the attribute a may take. When a further transaction relies on a precondition for a, the database checks whether the precondition evaluates to *true* for each value of a that is contained in the interval $[i, k]$. In contrast to the escrow locking technique, Bi-State-Termination, is a transaction termination mechanism. BST neither relies on numerical values, nor assumes that an attribute value must lie in a given interval. BST always knows the exact values that an attribute can actually have and even allows an application to decide that a transaction T_i may only be committed in a certain constellation of commit and abort decisions of transactions on which T_i depends.

Another related locking mechanism is Speculative Locking (SL) [59]. SL was proposed to speed up transaction processing by spawning multiple parallel executions of a transaction that waits for the acquisition of required locks. SL has in common with Bi-Sate-Termination that SL also allows a transaction T_c to access the after-image of a transaction T_i while T_i is waiting for its commit decision. However, unlike Bi-State-Termination, SL does not allow committing T_c before the final commit decision for T_i has been received. This means, SL cannot successfully terminate T_c while the commit vote for T_i is missing. For this reason, SL cannot be used to solve the infinite transaction blocking problem that may occur in mobile networks. Furthermore, our Fast-BST implementation can execute read-operations in one pass even if they return multiple result values due to transactions that wait for the commit decision.

Multiversion database systems [16, 18, 34] are used to support different expressions of a data object. They are used for CAD modelling and versioning systems.

However, compared to BST, multiversion database systems allow multiple versions to be concurrently valid, while BST allows only one valid version, but lacks the knowledge which of the multiple versions is valid due to the atomic commit protocol. Whenever BST requires multiple transaction executions that all return the same result, BST is even transparent to the application. Furthermore, multiversion database systems are mostly central embedded databases that are not designed to deal with distributed transactions. Instead, the user explicitly specifies on which version he wants to work.

Non-locking concurrency control like multiversion concurrency control [5, 71], timestamp-based concurrency control [44], or optimistic concurrency control [32,39] omit the use of locks. However, this does not solve the infinite transaction blocking problem on concurrent transactions, since the database proposes that it will commit the transaction by sending the voteCommit message, regardless of the used concurrency control mechanism. Therefore, without Bi-State-Termination, the database cannot process a transaction T_c that is dependent on a transaction T_i, while T_i waits for the final commit decision, even if the database uses locking-free concurrency control. This motivates the use of BST, which is a termination mechanism that supports the actually used concurrency control mechanism.

Even 1PC [1, 4], which does not require a vote message but acknowledges each operation, encounters the problem of transaction blocking since each acknowledged operation that accesses a data tuple must block this data tuple until the transaction is successfully completed.

6.8 Summary and Conclusion

We have shown that whenever an atomic commitment is necessary and an atomic commit protocol is used, transaction blocking occurs. Although the risk of protocol blocking can be minimized by using atomic commit protocols with multiple coordinators, the risk of infinite transaction blocking, which can occur if the database moves or disconnects, is not appropriately solved by current approaches. We have explained the concept of Bi-State-Termination, which is useful to terminate blocked transactions even without knowing the explicit coordinator decision and have described three implementations.

Our experimental results have shown that Bi-State-Termination enhances the number of committed transactions and that BST is able to deal with a large number of depending blocked transactions without experiencing significant performance

loss. This justifies using BST in mobile ad-hoc networks that are exposed to the risk of transaction blocking.

To summarize, we consider Bi-State-Termination as a useful option that is usable for mobile networks in order to terminate a transaction instead of just waiting for the commit decision for a long time.

Chapter 7

Summary and Conclusion

In this thesis, we have examined transaction processing in mobile ad-hoc networks and have mainly focussed on the reduction of blocking.

We have developed a Web service transaction model that is especially targeted on dynamic service invocation in mobile ad-hoc networks since the combination with the proposed Commit tree implicitly flattens the invocation hierarchy at commit time in order to speed up atomic commit protocol execution time. Furthermore, our Web service transaction model benefits from asynchronous service invocation as participants can invoke multiple services in parallel.

We have presented a technique that uses a special non-blocking state – the "Adjourn state" – to eliminate the need of setting up participant time-outs for aborting a transaction. This allows the Adjourn state to balance resource allocation on demand, since locks for resources that are frequently accessed are released and transferred to other transactions at an early stage – in contrast to resources that are only required by a single transaction. In combination with our Web service transaction model, the Adjourn state allows a flexible reaction to network failures to make renewed invocations of sub-transactions in many cases superfluous, for instance when a sub-transaction must be re-invoked with the same parameters and has been aborted due to concurrency issues.

We have shown that the Adjourn state reduces transaction blocking even before the atomic commit protocol starts and that the Adjourn state enhances the transaction throughput in networks where communication links between two participants often break. Furthermore, our experiments have shown that the more unreliable a mobile network is, the greater the benefit that transaction throughput and transaction blocking undergo by using the Adjourn state concurrency control enhancement compared to traditional approaches. This motivates the use of the Adjourn state in mobile ad-hoc networks in combination with locking and validation-based concurrency control mechanisms.

Blocking also occurs during the execution of the atomic commit protocol which may become critical when voteCommit messages or doCommit messages are delayed or lost. Therefore, we have developed the atomic commit protocol CLCP, which employs multiple coordinators and thus enhances the protocol availability. In contrast to existing atomic commit protocols, CLCP is a decentralized protocol that lets participants determine the transaction's decision by their own knowledge, making messages that inform participants of the transaction's decision superfluous. Furthermore, CLCP uses a timeout mechanism that allows each participant to speculate on another participant's failure by sharing its own knowledge about participants which might have disappeared. While traditional atomic commit protocols solely operate on the application layer, the cross-layer design of CLCP makes the use of acknowledgement messages superfluous and allows CLCP to save energy.

We have proven the correctness of the two phases of CLCP including the liveness and correctness properties that are crucial for atomic commit protocols.

Our experiments have demonstrated that CLCP significantly reduces the average blocking duration and that CLCP's energy consumption is remarkably less than that of other consensus-based protocols and even comparable to the energy consumption of protocols that do not use acknowledgements. As our experiments have shown, CLCP is superior to all the other protocols regarding the number of committed transactions.

We have explained that in the seldom event that no majority of transaction participants exists within one partition, no atomic commit protocol is free of blocking. To handle this case and to handle the case that a participant does not receive the coordinator's commit decision for a long period of time, we have proposed the use of Bi-State-Termination. This technique allows continuing the processing of transactions T_i, even in the case that transactions T_o are blocked, by obeying both possible outcomes of the blocked transactions T_o. This allows the transactions T_i to commit even if they conflict with pending transactions T_o.

In our experiments, we have compared three implementations of Bi-State-Termination. The Fast-BST implementation, which adds the conditions under which each data tuple becomes valid directly to each tuple, has proven to be the fastest implementation. The experimental evaluation of the Fast-BST implementation in the TPC-C benchmark, which captures a real world scenario with a high transaction load, has proven that Bi-State-Termination-enabled transaction processing allows to commit significantly more transactions than traditional transaction processing.

This highly motivates the use of Bi-State-Termination in mobile networks, where network failures due to participant movement are likely.

A challenging topic for further research in transaction processing in mobile networks beyond this thesis is the investigation of persistence. For fixed-wired networks, a lot of research has been contributed. Today, only hardware failures or physical disasters can violate persistency in fixed-wired networks. In mobile ad-hoc networks, however, the movement of participants with the resulting loss of communication and the limitations of battery power add a new dimension to research on persistence. Furthermore, participants can dynamically cooperate to guarantee persistence within a predefined geographical area.

Transaction processing would highly benefit from solid persistence layers in order to archive the transaction's commit decision within the network. Thus, participants that have moved during the atomic commit protocol execution and could not be informed about the transaction decision could use the persistence layer to lookup the transaction decision. Thus, an interesting orthogonal extension of research in mobile networks is to focus on distributed persistence strategies for commit decisions that involve additional mobile nodes.

To summarize, atomicity, serializability, and data consistency are highly desired transactional guarantees not only in fixed-wired networks, but also in mobile ad-hoc networks. Transaction blocking has been the major reason why many contributions for mobile ad-hoc networks have relaxed their transactional guarantees. In order to reduce blocking while still guaranteeing atomicity, serializability, and data consistency, we have introduced the Adjourn state and have presented two Adjourn state implementations for both concurrency control schemes, locking and validation. Furthermore, in order to reduce blocking caused by atomic commit protocols, we have presented CLCP and Bi-State-Termination, both of which are orthogonal to the concurrency control schema. Therefore, using our techniques, blocking can be considered significantly less harmful to transactions in mobile ad-hoc networks.

Bibliography

[1] M. Abdallah, R. Guerraoui, and P. Pucheral. One-phase commit: Does it make sense? In *ICPADS '98: Proceedings of the 1998 International Conference on Parallel and Distributed Systems*, page 182, Washington, DC, USA, 1998. IEEE Computer Society.

[2] Y. Al-Houmaily, P. K. Chrysanthis, and S. P. Levitan. An argument in favor of the presumed commit protocol. In *Proc. of the 13th International Conference on Data Engineering (ICDE)*, pages 255–265, April 1997.

[3] Y. J. Al-Houmaily and P. K. Chrysanthis. The implicit-yes vote commit protocol with delegation of commitment. In *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems*, pages 804–810, September 1996.

[4] Y. J. Al-Houmaily and P. K. Chrysanthis. 1-2pc: the one-two phase atomic commit protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17*, pages 684–691, 2004.

[5] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

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

[7] J.-H. Böse, S. Böttcher, P. K. Chrysanthis, A. Delis, L. Gruenwald, A. Mondal, S. Obermeier, A. Ouksel, G. Samaras, and S. Viglas. 06431 working group summary: Atomicity in mobile networks. In S. Böttcher, L. Gruenwald, P. J. Marrón, and E. Pitoura, editors, *Scalable Data Management in Evolving Networks*, number 06431 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[8] S. Böttcher, L. Gruenwald, and S. Obermeier. Reducing sub-transaction aborts and blocking time within atomic commit protocols. In *23rd British National*

Conference on Databases (BNCOD), Belfast, Northern Ireland, UK, pages 59–72, 2006.

[9] S. Böttcher, L. Gruenwald, and S. Obermeier. A failure tolerating atomic commit protocol for mobile environments. In *Proceedings of the The 8th International Conference on Mobile Data Management (MDM 2007), Mannheim, Germany*, 2007.

[10] S. Böttcher and S. Obermeier. Dynamic commit tree management for service oriented architectures. In *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS), Funchal, Madeira - Portugal*, 2007.

[11] S. Böttcher, S. Obermeier, A. Türling, and J. H. Wiesner. Segmentation-based caching for mobile auctions. In *Proceedings of the International Conference on Techniques and Applications for Mobile Commerce (TAMOCO 2008), Glasgow, Great Britain*, 2008.

[12] J. Böse, S. Böttcher, L. Gruenwald, S. Obermeier, H. Schweppe, and T. Steenweg. An integrated commit protocol for mobile network databases. In *9th International Database Engineering & Application Symposium IDEAS*, Montreal, Canada, 2005.

[13] S. Böttcher, L. Gruenwald, and S. Obermeier. An atomic web-service transaction protocol for mobile environments. In *2nd International Workshop on Pervasive Information Management (PIM 2006)*, Munich, Germany, 2006. to appear.

[14] L. F. Cabrera, G. Copeland, M. Feingold, et al. Web Services Transactions specifications – Web Services Atomic Transaction. `http://www-128.ibm.com/developerworks/library/specification/ws-tx/`, 2005.

[15] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2(5):483–502, 2002.

[16] W. Cellary and G. Jomier. Consistency of versions in object-oriented databases. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 432–441. Morgan Kaufmann, 1990.

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

[18] I.-M. A. Chen, V. M. Markowitz, S. Letovsky, P. Li, and K. H. Fasman. Version management for scientific databases. In P. M. G. Apers, M. Bouzeghoub,

and G. Gardarin, editors, *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 1996.

[19] S. Covaci, T. Zhang, and I. Busse. Java-based intelligent mobile agents for open system management. In *ICTAI '97: Proceedings of the 9th International Conference on Tools with Artificial Intelligence (ICTAI '97)*, page 492, Washington, DC, USA, 1997. IEEE Computer Society.

[20] F. Curbera, Y. Goland, J. Klein, F. Leymann, et al. Business Process Execution Language for Web Services, V1.0. Technical report, BEA, IBM, Microsoft, 2002-07-31 2002.

[21] R. A. Dirckze and L. Gruenwald. A toggle transaction management technique for mobile multidatabases. In *CIKM '98*, pages 371–377, New York, USA, 1998. ACM Press.

[22] R. A. Dirckze and L. Gruenwald. A pre-serialization transaction management technique for mobile multidatabases. *Mobile Networks and Applications*, 5(4):311–321, 2000.

[23] M. H. Dunham, A. Helal, and S. Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *Mobile Networks and Applications*, 2(2):149–162, 1997.

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

[25] L. M. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *IEEE INFOCOM*, 2001.

[26] A. Gallais, H. Parvery, J. Carle, J.-M. Gorce, and D. Simplot-Ryl. Efficiency impairment of wireless sensor networks protocols under realistic physical layer conditions. In *Proc. 10th IEEE International Conference on Communication Systems (ICCS 2006)*, Singapore, 2006.

[27] S. Giordano, I. Stojmenovic, and L. Blazevie. Position based routing algorithms for ad hoc networks: a taxonomy. `http://www.site.uottawa.ca/~ivan/routing-survey.pdf`, 2001.

[28] J. Gray. Notes on data base operating systems. In M. J. Flynn, J. Gray, A. K. Jones, et al., editors, *Advanced Course: Operating Systems*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer, 1978.

[29] J. Gray. A comparison of the byzantine agreement problem and the transaction commit problem. In *Fault-tolerant distributed computing*, pages 10–17, London, UK, 1990. Springer-Verlag.

[30] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

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

[32] T. Haerder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, 1984.

[33] T. Hou and V. Li. Transmission range control in multihop packet radio networks. *IEEE Transactions on Communications*, 34(1):38–44, 1986.

[34] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–409, 1990.

[35] M. Kifer, A. Bernstein, and P. M. Lewis. *Database Systems: An Application Oriented Approach.* Pearson Addison-Wesley, 2005.

[36] W. Kohler, A. Shah, and F. Raab. Overview of TPC Benchmark C: The Order-Entry Benchmark. Technical report, `http://www.tpc.org`, Transaction Processing Performance Council, 1991.

[37] V. Kumar and M. Hsu. *Recovery mechanisms in database systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[38] V. Kumar, N. Prabhu, M. H. Dunham, and A. Y. Seydim. Tcot - a timeout-based mobile transaction commitment protocol. *IEEE Transactions on Computers*, 51(10):1212–1218, 2002.

[39] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[40] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[41] L. Lamport and M. Massa. Cheap paxos. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 307, Washington, DC, USA, 2004. IEEE Computer Society.

[42] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. In *Proc. Intl' Conf. on Very Large Data Bases*, page 630, Dublin, Ireland, Aug. 1993.

[43] V. C. S. Lee, K.-W. Lam, S. H. Son, and E. Y. M. Chan. On transaction processing with partial validation and timestamp ordering in mobile broadcast environments. *IEEE Trans. Comput.*, 51(10):1196–1211, 2002.

[44] P.-J. Leu and B. K. Bhargava. Multidimensional timestamp protocols for concurrency control. In *Proceedings of the Second International Conference on Data Engineering*, pages 482–489, Washington, DC, USA, 1986. IEEE Computer Society.

[45] C. Liebig and A. Kühne. Open Source Implementation of the CORBA Object Transaction Service. http://xots.sourceforge.net/, 2005.

[46] B. Lindsay, P. Selinger, C. Galtieri, J. Gray, R. Lorie, T. Price, F. Putzolo, I. Traiger, and B. Wade. Notes on distributed databases. *IBM Almaden Res. Lab., Technical Report RJ2571*, 1979.

[47] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 76–88, New York, NY, USA, 1983. ACM Press.

[48] C. Mohan, B. Lindsay, and R. Obermark. Transaction management in the R* distributed database management system. *ACM Trans. on Database Sys.*, 11(4):378, Dec. 1986.

[49] N. Nouali, A. Doucet, and H. Drias. A two-phase commit protocol for mobile wireless environment. In H. E. Williams and G. Dobbie, editors, *Sixteenth Australasian Database Conference (ADC2005)*, volume 39 of *CRPIT*, pages 135–144, Newcastle, Australia, 2005. ACS.

[50] S. Obermeier and S. Böttcher. Avoiding infinite blocking of mobile transactions. In *Proceedings of the 11th International Database Engineering & Applications Symposium (IDEAS), Banff, Canada*, 2007.

[51] S. Obermeier, S. Böttcher, M. Hett, P. K. Chrysanthis, and G. Samaras. Adjourn state concurrency control avoiding time-out problems in atomic commit protocols (poster). In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE), Cancun, Mexico*, 2008.

[52] S. Obermeier, S. Böttcher, M. Hett, P. K. Chrysanthis, and G. Samaras. Blocking reduction for distributed transaction processing within manets. In *submitted for publication*, 2008.

[53] S. Obermeier, S. Böttcher, and D. Kleine. CLCP – a distributed cross-layer commit protocol for mobile ad-hoc networks. In *submitted for publication*, 2008.

[54] S. Obermeier, S. Böttcher, and T. Wycisk. Xpath selectivity estimation for a mobile auction application. In *Proceedings of the 11th International Database Engineering & Applications Symposium (IDEAS), Banff, Canada*, 2007.

[55] Object Management Group. Transaction Service Specification 1.4. `http://www.omg.org`, 2003.

[56] E. Pitoura and B. K. Bhargava. Maintaining consistency of data in mobile distributed environments. In *Intl. Conf. on Distributed Computing Systems*, pages 404–413, 1995.

[57] A. Rakotonirainy. Adaptable transaction consistency for mobile environments. In *DEXA Workshop*, pages 440–445, 1998.

[58] P. K. Reddy and M. Kitsuregawa. Reducing the blocking in two-phase commit with backup sites. *Inf. Process. Lett.*, 86(1):39–47, 2003.

[59] P. K. Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):154–169, 2004.

[60] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations in a commercial distributed environment. *Distrib. Parallel Databases*, 3(4):325–360, 1995.

[61] P. Shigiltchoff and E. Chrysanthis. Multiversion data broadcast organizations. In *Proceedings of the 6th East-European Conference on Advances in Databases and Information Systems*, 2002.

[62] D. Skeen. Nonblocking commit protocols. In Y. E. Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan*, pages 133–142. ACM Press, 1981.

[63] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. In *Berkeley Workshop*, pages 129–142, 1981.

[64] P. M. Spiro, A. M. Joshi, and T. K. Rengarajan. Designing an optimized transaction committ protocol. *Digital Technical Journal*, 3(1):0–, 1991.

[65] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. on Softw. Eng.*, 5(3), May 1979.

[66] M. R. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. In *Distributed systems, Vol. II: distributed data base systems*, pages 193–199, Norwood, MA, USA, 1986. Artech House, Inc.

[67] S. Thatte. XLANG - Web Services for Business Process Design. Initial public draft, Microsoft Corporation, 2002-03-22 2001.

[68] J. D. Ullman. *Principles of Database Systems, 2nd Edition*. Computer Science Press, 1982.

[69] P. Urban, N. Hayashibara, A. Schiper, and T. Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *SRDS*

'04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04), pages 4–17, Washington, DC, USA, 2004. IEEE Computer Society.

[70] C. Waal and M. Gerharz. BonnMotion: a mobility scenario generation and analysis tool. University of Bonn, Germany, `http://web.informatik.uni-bonn.de/IV/Mitarbeiter/dewaal/BonnMotion`, 2003.

[71] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[72] D. Y. Ye, M. C. Lee, and T. I. Wang. Mobile agents for distributed transactions of a distributed heterogeneous database system. In *DEXA 02*, pages 403–412, London, UK, 2002. Springer-Verlag.