

# Efficient Commit Processing in High-Availability Main-Memory Databases

Heine Kolltveit

May 13, 2008

# Abstract

Database systems are currently being used as backbone to thousands of applications. Some of these have very high demands for high availability and fast real-time responses. Examples include telecommunication systems, trading systems, online gaming and sensor networks.

Disk-based databases are too slow to facilitate the short response time requirements of these types of systems. Main-memory databases, however, provide short response times and, if replicated, the required level of availability. Still, the execution of transactions should be optimized to support as many transactions per second as possible while keeping the response times at acceptable levels.

The focus of this work is how to improve the performance of main-memory primary-backup systems by looking at the execution of transactions and commit protocols. Furthermore, system resilience to failures is addressed. We have examined existing solutions both for distributed environments in general and main-memory primary-backup systems in particular.

The main contribution of this thesis deals with protocols and several optimizations for committing transactions in replicated main-memory databases. The protocols reduce the overhead by sending messages in a circular fashion. In addition, the results of an extensive investigation of existing commit protocols and optimizations for them are presented as related work. Furthermore, the protocols are extended to multiple copies and an approach to handle failures in non-deterministic environments. This improves the resilience to failures.

The suggested protocols and optimizations have been evaluated. The results show a 50% increase in throughput and a reduction in transaction response time are achievable compared to state-of-the-art protocols. Throughput for single-tuple transactions can be further improved by 80% – 130%, depending on the ratio between reads and writes. Hence, the work clearly has merit for distributed databases with high availability and short real-time response requirements.



# Preface

This thesis is submitted to the Norwegian University of Science and Technology in partial fulfillment of the degree PhD. The work has been carried out at the Database System Group, Department of Computer and Information Science (IDI). The study was funded by the Faculty of Information Technology, Mathematics and Electrical Engineering through the “forskernskolen” program. Some minor grammatical and spelling errors have been corrected in the papers in Part II compared to their published versions.

## Acknowledgement

First of all, I would like to thank my supervisor Professor Svein-Olaf Hvasshovd for his advice, collaboration, and ideas, and for providing valuable comments to drafts of the thesis and papers. I would also like to thank my co-advisors Dr. Ing. Øystein Torbjønsen and Professor Svein Erik Bratsberg for their guidance and valuable feedback during this work.

Moreover, I’ve received comments and help from various sources during the years I’ve been working on my thesis. I was sharing office with Jørgen Løland and enjoyed many motivating conversations with him. Together with the rest of the lunch team; Jeanine Lilleng, Jon Olav Hauglid, Norvald Ryeng and others, we have enjoyed many stimulating discussions on subjects both on and off topic.

Kjetil Nørvåg has been helpful commenting on my papers and taught me much about scientific publishing. I thank the administrative staff at IDI and the Database System Group in general for providing a good scientific environment where I could carry out my research.

Finally, I would like to thank my brother, Njal Karevoll, for discussing my ideas and helping me evolve, and my parents for always supporting me.



# Contents

<b>I</b>	<b>Background and Context</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.1.1	The Challenges of Main-Memory Commit Processing . . . . .	10
1.2	Research Questions . . . . .	11
1.3	Research Methodology . . . . .	12
1.4	Thesis Organization . . . . .	13
<b>2</b>	<b>Fault Tolerance in Distributed Systems</b>	<b>15</b>
2.1	Dependability and Fault Tolerance . . . . .	15
2.1.1	Terminology . . . . .	15
2.1.2	Module Reliability and Availability . . . . .	16
2.1.3	Failure Classification . . . . .	16
2.1.4	Failure Semantics . . . . .	17
2.1.5	Failure Masking . . . . .	18
2.1.6	Liveness and Safety . . . . .	18
2.2	Distributed Systems . . . . .	19
2.2.1	Characteristics of Distributed Systems . . . . .	19
2.2.2	Distributed Applications . . . . .	19
2.2.3	Real-Time Systems . . . . .	20
2.3	Replication in Distributed Systems . . . . .	20
2.3.1	Group Communication . . . . .	21
2.3.2	Replica Consistency . . . . .	24
2.3.3	Replica Determinism . . . . .	24
2.3.4	Replication Strategies . . . . .	25
2.3.5	Orphan Requests . . . . .	29
2.4	Transactions . . . . .	30
2.4.1	Properties . . . . .	30
2.4.2	Concurrency Control . . . . .	31
2.4.3	Recovery . . . . .	32
2.4.4	Transactional Systems Architectures . . . . .	34

2.4.5	Distributed Transaction Entities . . . . .	34
2.4.6	Atomic Commitment . . . . .	35
<b>3</b>	<b>Survey of Commit Processing</b>	<b>37</b>
3.1	Commit Processing: Blocking Approaches . . . . .	37
3.1.1	Two-Phase Commit . . . . .	37
3.1.2	Presumed Abort . . . . .	38
3.1.3	Presumed Commit . . . . .	39
3.1.4	Other Optimizations . . . . .	40
3.1.5	One-phased Commit Protocols . . . . .	42
3.1.6	Optimistic Protocols . . . . .	45
3.2	Commit Processing: Non-Blocking Approaches . . . . .	46
3.2.1	Extra Round of Messages . . . . .	46
3.2.2	Broadcast and Multicast . . . . .	47
3.2.3	Replicated Coordinator . . . . .	49
3.2.4	Other Non-Blocking Approaches . . . . .	53
3.3	Managing Non-Determinism in Replicated Transactional Systems . . . . .	54
<b>4</b>	<b>Conclusions</b>	<b>57</b>
4.1	Contributions of this thesis . . . . .	57
4.2	Limitations . . . . .	59
4.3	Future Work . . . . .	59
	<b>Bibliography</b>	<b>60</b>
<b>II</b>	<b>Publications</b>	<b>73</b>
<b>I</b>	<b>Preventing Orphan Requests by Integrating Replication and Transactions</b>	<b>75</b>
I.1	Introduction . . . . .	77
I.2	System Model . . . . .	79
I.3	Related Work . . . . .	80
I.4	Integration of Transactions and Replication . . . . .	81
I.4.1	Replicating the Transaction Manager . . . . .	81
I.4.2	Replicating the Transaction Participants . . . . .	82
I.4.3	Transaction Termination in Failure Scenarios . . . . .	85
I.5	Implementation and Testing . . . . .	86

---

I.5.1	Prototype Implementation . . . . .	86
I.5.2	The Test Environment . . . . .	86
I.6	Comparing the Test Results . . . . .	88
I.6.1	Cost of Replication . . . . .	89
I.6.2	Fail-over Delay . . . . .	90
I.7	Conclusion and Further Work . . . . .	90
<b>II</b>	<b>The Circular Two-Phase Commit Protocol</b>	<b>95</b>
II.1	Introduction . . . . .	97
II.2	Related Work . . . . .	98
II.3	System Model . . . . .	100
II.4	The Non-Blocking Atomic Commit Problem . . . . .	101
II.5	The Circular Two-Phase Commit Protocol . . . . .	101
II.5.1	Detailed Description . . . . .	104
II.5.2	Correctness . . . . .	107
II.5.3	C1PC . . . . .	110
II.6	Evaluation . . . . .	111
II.7	Conclusion . . . . .	112
<b>III</b>	<b>Efficient High-Availability Commit Processing</b>	<b>117</b>
III.1	Introduction . . . . .	119
III.2	Related Work . . . . .	121
III.2.1	Optimizations . . . . .	121
III.2.2	Performance evaluations . . . . .	122
III.3	Commit Protocols . . . . .	123
III.3.1	Two-Phase Commit . . . . .	123
III.3.2	Replicated Two-Phase Commit . . . . .	123
III.3.3	Circular Two-Phase Commit . . . . .	124
III.3.4	One-Phased Protocols . . . . .	125
III.4	Dynamic Coordinators . . . . .	126
III.4.1	Dynamic Backup Coordinator . . . . .	126
III.4.2	Dynamic Primary Coordinator . . . . .	127
III.4.3	Dynamic Primary Coordinator using Piggybacking . . . . .	128
III.5	The Simulation Model . . . . .	128
III.6	Simulation Results . . . . .	131
III.6.1	Average Response Time . . . . .	131
III.6.2	Response Time Demands . . . . .	133
III.6.3	Distribution . . . . .	134
III.7	Analysis . . . . .	134

III.8 Evaluation and Conclusion . . . . .	136
<b>IV Main-Memory Commit Processing: The Impact of Priorities</b>	<b>141</b>
IV.1 Introduction . . . . .	143
IV.2 Related Work . . . . .	144
IV.3 Priorities and Piggybacking . . . . .	145
IV.3.1 Priority of Tasks by the Transaction Manager . . . . .	145
IV.3.2 Priority and Piggybacking Messages over the Network . . . . .	146
IV.4 The Simulation Model . . . . .	147
IV.5 The Simulation Results . . . . .	147
IV.5.1 Average Response Time . . . . .	147
IV.5.2 Response Time Demands . . . . .	148
IV.6 Analysis . . . . .	150
IV.7 Evaluation and Conclusion . . . . .	151
<b>V Main-Memory Commit Protocols for Multiple Backups</b>	<b>155</b>
V.1 Introduction . . . . .	157
V.2 Related Work . . . . .	158
V.3 Main-Memory Commit Protocols . . . . .	159
V.3.1 R2PC . . . . .	159
V.3.2 C2PC . . . . .	159
V.3.3 R1PC . . . . .	160
V.3.4 C1PC . . . . .	161
V.4 Extending to Multiple Backups . . . . .	161
V.4.1 R2PC . . . . .	161
V.4.2 C2PC . . . . .	161
V.4.3 RC2PC . . . . .	162
V.4.4 One-Phased Multiple Backups Protocols . . . . .	163
V.5 The Simulation Model . . . . .	163
V.6 Simulation Results . . . . .	163
V.6.1 Average Response Time . . . . .	164
V.6.2 Response Time Demands . . . . .	165
V.7 Analysis . . . . .	167
V.8 Evaluation and Conclusion . . . . .	168
<b>VI Efficient Execution of Small (Single-Tuple) Transactions in Main-Memory Databases</b>	<b>173</b>
VI.1 Introduction . . . . .	175
VI.2 Related Work . . . . .	177
VI.2.1 Transaction Type . . . . .	177
VI.2.2 Main-Memory Commit Processing . . . . .	178

---

VI.3 Using The Semantics of Read and Write Operations . . . . .	180
VI.3.1 Simple Update . . . . .	180
VI.3.2 Simple Query . . . . .	181
VI.4 Correctness . . . . .	182
VI.4.1 The Non-Blocking Atomic Commitment Problem . . .	182
VI.4.2 Proof . . . . .	183
VI.5 The Simulation Model . . . . .	185
VI.6 Simulation Experiments . . . . .	188
VI.6.1 Average Response Time . . . . .	188
VI.6.2 Response Time Demands . . . . .	189
VI.7 Analysis . . . . .	191
VI.8 Conclusion and Further Work . . . . .	192



# List of Figures

1.1	The Home Location Register in mobile networks . . . . .	7
2.1	Active replication . . . . .	26
2.2	Passive replication . . . . .	27
2.3	Semi-passive replication . . . . .	28
2.4	A non-deterministic passive server causing a replicated invocation . . . . .	29
2.5	An example of an orphan request . . . . .	30
3.1	The two-phase commit protocol . . . . .	38
3.2	The two-phase presumed abort protocol, abort case . . . . .	39
3.3	The two-phase presumed commit protocol, commit case . . . . .	40
3.4	The two-phase protocol with linear commit optimization . . . . .	40
3.5	The two-phase protocol with read-only optimization . . . . .	41
3.6	The Unsolicited Vote approach . . . . .	43
3.7	The Early Prepare approach . . . . .	44
3.8	The Coordinator Log approach . . . . .	44
3.9	The Implicit Yes-Vote approach . . . . .	46
3.10	The Three-Phase Commit protocol . . . . .	47
3.11	The SPAC and NB-SPAC protocols . . . . .	48
3.12	The Backup Commit Protocol . . . . .	50
3.13	The ClustRa Commit protocol . . . . .	51
3.14	The 2-safe Primary-Backup Commit Protocol . . . . .	52
3.15	The MySQL Cluster Commit protocol . . . . .	53
I.1	An orphan request caused by non-determinism in server <i>A</i> . . . . .	78
I.2	The execution and join phase of a transaction . . . . .	82
I.3	The successful termination of a transaction . . . . .	82
I.4	A failure of a primary, and the consequent fail-over . . . . .	83
I.5	A failure of a primary, without the fail-over . . . . .	84
I.6	A double fail-over . . . . .	84

I.7	A checkpoint and a possible orphan request in service $B$ . . . . .	85
I.8	A model of the system used for testing . . . . .	87
II.1	Execution of various atomic commitment protocols . . . . .	102
II.2	Examples of C1PC execution . . . . .	110
II.3	Delays and total overhead for various ACPs . . . . .	111
III.1	Legend for Fig. III.2 and Fig. III.3 . . . . .	124
III.2	Main-memory commit protocols . . . . .	124
III.3	Execution of C2PC with optimizations . . . . .	127
III.4	The logical model of the system . . . . .	129
III.5	Average response times using dynamic coordinators . . . . .	132
III.6	95% maximum response time for C1PC based protocols . . . . .	133
III.7	Distribution of response times for C1PC and DPC-P . . . . .	134
III.8	Comparing analysis and simulations for all protocols using DPC-P . . . . .	135
IV.1	Changes in response time by using piggybacking and priorities	149
IV.2	95% maximum response time using piggybacking and priorities	150
IV.3	Comparison of analysis and simulations . . . . .	150
V.1	Main-memory commit protocols for one backup . . . . .	160
V.2	Commit protocols for multiple backups, all executed with EA	162
V.3	Two-phased protocols executed with EA . . . . .	165
V.4	One-phased protocols executed with EA . . . . .	166
V.5	Two-phased protocols using the EA optimization . . . . .	167
V.6	One-phased protocols using the EA optimization . . . . .	168
V.7	Comparison of analysis and simulations . . . . .	168
VI.1	Existing commit processing approaches and optimizations for main memory . . . . .	179
VI.2	The protocol for single-update . . . . .	181
VI.3	Logical model of the system . . . . .	185
VI.4	The average response time. . . . .	189
VI.5	The maximum response time for the 95% quickest transactions	190
VI.6	Comparison between analyzes and simulations of SUSR . . . . .	191

# List of Tables

I.1	A summary of the response times for the test runs in Section I.5.2 . . . . .	88
III.1	The input parameters of the simulation . . . . .	130
IV.1	The input parameters of the simulation . . . . .	148
V.1	The input parameters of the simulation . . . . .	164
VI.1	The input parameters of the simulation . . . . .	186



# Part I

## Background and Context



# Chapter 1

## Introduction

The topic of this thesis is commit processing in replicated main-memory database systems with shared-nothing architecture. The focus is on how performance of main-memory commit processing can be improved and how to manage the effects of non-determinism while incurring minimum performance degradation. High availability and consistency must not be jeopardized.

This chapter first presents the motivation and the objectives for our work, and then discusses the research questions and methodology.

### 1.1 Motivation

Currently, database systems are used to store information for a wide variety of applications. These applications range from small personal archives with contact information for friends and family to complex commercial database systems storing huge amounts of mission critical and sensitive data. Naturally, the requirements for databases differ as well. For instance, the user could happily be waiting several seconds before getting a contact's telephone number, or if it is unavailable, to search for it using the white or yellow pages. Unavailability or slow response in databases used by business applications could, however, be financially devastating and, in worst case, cause injuries or deaths.

High availability and real-time responses are not required for all business applications, but many systems do. Examples include real-time process control systems, telephone switching networks, sensor networks, trading systems, embedded systems and electronic toll collecting systems. Typically, these systems generate a very large transaction workload against the database, and a large part of the workload consists of short read and update transactions.

As part of the ACID properties [44], the database must ensure that the

transaction is completed as an *atomic operation* or not at all. In addition, the outcome of the transaction, including caused state changes, must be *persistently* stored before the new data is made available to the outside world or other transactions. This is called *commit processing*. Commit processing should add as little overhead as possible to transaction processing. The standard commit processing protocol is the 2-phase commit protocol (2PC), which requires two rounds of messages and two disk writes per cohort. For real-time systems, however, the overhead of the industry standard 2PC is too large. Existing research has mostly focused on reducing the number of disk writes or communication steps, and the very few approaches that avoid disk writes are not very efficient [55].

The rest of this section presents the main-memory database, which provides the setting for this thesis as it has the potential of fulfilling high availability and real-time response requirements. Main memory is then classified and precompiled transactions are presented. Finally, motivating examples of existing and emerging applications for the results of the work in this thesis are given.

### Main-Memory Databases

Traditional databases are disk-based [39, 11]. To be stored *persistently*, data and log must be written to disk. This ensures data can be restored after a system crash. However, read and write accesses to disk takes time, which led to *main-memory databases* or MMDBs [33]. The advantage of using main memory is that reading and writing is very fast. Measurements presented in [9], shows that main-memory databases can outperform commercial disk-based systems by a factor of 40 for a simple lookup for a phone number. The disadvantages are that main-memory size is limited and data in it is *volatile*; i.e., data is lost after a crash. However, by connecting many nodes together using a communication network, these disadvantages can be reduced. The possible size of the database becomes the total size of main memory in all machines in the distributed system. In addition, data can be replicated across nodes, ensuring persistence and availability: If one node goes down, the other takes over the processing of transactions.

Many commercial and research MMDBs have been developed: Oracle TimesTen [106] (earlier Smallbase [48]), MySQL Cluster [77], GigaSpaces [36], ClustRa [50], Polyhedra [87], DataBlitz [9] (earlier Dalí [53]), solidDB [100], FastDB [28], MonetDB [46], RODAIN [65], etc. For an MMDB to reach its full performance potential, all parts of the transaction processing and data storage must be tailored to main-memory environments. This thesis concentrates on the final part of transaction processing, where the outcome

for each transaction is decided, namely the commit processing.

### MMDB Classification

MMDBs can be classified according to their access methods, log policy and persistence policy. For the first, since data resides in main memory, special access methods yield better results than traditional ones. The second governs whether the log is saved to disk or not. In MMDBs, data can be kept in main memory, while the log is written to disk to enable disk-based recovery after a crash, or both the log and data reside in main memory only. The third determines if the log is persistently stored or not. If it is written to disk, it is persistently stored if *Write Ahead Logging* is used; i.e., the log is on disk before the data. If not, the database cannot be guaranteed to be recoverable. In *pure* main-memory systems, both log and data are saved in main memory only. It can, however, be made persistent by using *replication* between processes at different physical locations.

This thesis is in the context of a shared-nothing pure main-memory system where both data and log are kept in main memory only and replication is used for persistence. This setting provides very good support for databases with high availability requirements and soft real-time deadlines, where a high percentage of transactions should be completed within a short timeframe.

With such response time requirements, precompiled transactions or *persistent stored modules* are needed. These are presented below.

**Persistent Store Modules** Standard SQL queries often have multiple request-reply interactions before a transaction is committed. The interactions add extra communication time, and if a user needs to react to participate in the interaction, even more time is added. Stored procedures or *Persistent Store Modules* [52], however, is a sort of precompiled transaction which is compiled and stored in the database. Later, it is called with possible parameters, and the client receives the results. These transactions are important in real-time environments with short deadlines.

### General Profile And Motivating Examples

This work is focused on improving the performance and resilience of system which has these general properties and requirements:

- Real-time environments
- Short update transactions are a large part of the load

- Clients require very short response times
- The updates must be readily available to the read transactions
- Transactional consistency – the consistency should not be relaxed
- High availability

This rest of this section presents some motivating examples of existing and emerging applications which exhibit these properties.

**Home Location Register** The Home Location Register (HLR) is the central database in any GSM (Global System for Mobile communications) network. Its requirements are [94]:

- It must be able to handle thousands of updates and reads per second generated by the millions of subscribers, i.e. high throughput.
- The updates and reads must happen in real-time and have response time within a few tens of milliseconds.
- It must have high availability corresponding to a maximum unavailability of  $10^{-6}$ , since any unavailability cause the entire network to be unavailable.
- It must be scalable, facilitating an increase in the subscription base.

These requirements are achievable by MMDBs. Existing MMDBs like Oracle TimesTen, MySQL Cluster and solidDB advertise being used for telephony switching networks [106, 77, 100]. The mobile industry is huge and growing. There are currently over 3.3 billion subscribers worldwide [92], and the world's largest mobile network operator (China Mobile) had over 356 million subscribers in China and Pakistan in October 2007 [74]. The first nine months of 2007, the total revenue for China Mobile was USD 35 billion [15]. A smaller example is Telenor Mobile Norway. It has 2.8 million subscribers by third quarter 2007 and total revenue of USD 2 billion in 2006. The potential financial losses for a mobile network operator, if the system does not fill the requirements, are huge. For instance, without the HLR, no mobile devices belonging to the network can be contacted.

GSM networks use a two-level location system. As shown in Figure 1.1, the HLR is the top-level. Its main purpose is to manage the mobility of mobile devices. It contains information about all mobile devices currently authorized to use the network or *subscribers*. The information covers the details of every valid SIM card the operator has issued and they include a

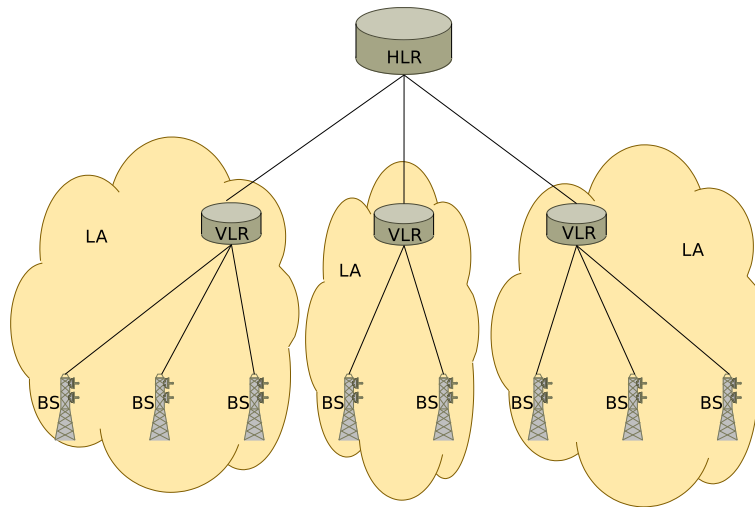


Figure 1.1: The Home Location Register in mobile networks

unique identifier for each SIM card, the telephone number(s) of the SIM card, GSM services, GPRS settings, call divert settings and the current location of the subscriber. There is generally one HLR per mobile network, although numbers can be partitioned across multiple database partitions. Data in HLR is stored until the subscriber leaves the operator.

A *Visiting Location Register* or VLR is the second-level database, which temporarily stores information about local subscribers in a region. These local subscribers are the devices that have roamed into the area of a *Base Station*, or BS, the VLR serves. The information is stored in the VLR until the subscriber moves to a Location Area (LA) served by another VLR or has been out of reach for a predetermined period of time. Each BS is connected to only one VLR, and each VLR is connected to only one HLR, as shown in Figure 1.1. An LA potentially includes tens or hundreds of BSs.

A HLR may manage tens of millions of mobile devices. When a device moves from one BS to another, the system must update the location of the device. Movements of mobile phones within the same LA are handled by the VLR. If a device moves to a BS connected to another VLR, the HLR manages the movement. Using assumptions from [72], each subscriber generates approximately one HLR update transaction per hour. Thus, 10 million subscribers generate a total load of around 2700 update transactions per second. These transactions require from 10 to 20 ms response time [50] to avoid glitches in calls when the device moves from one LA to another.

When a mobile device tries to make a call to another mobile device, the VLR is queried to see if the outgoing call is allowed. If it is allowed,

the HLR is used to lookup the VLR that is currently associated with the receiving device. If an associated VLR is found and no call forwarding has been issued, the VLR is contacted for the exact location of the device, and the call is routed through the appropriate BS. This process is called *call delivery*.

Each called delivered to a mobile device currently subscribing to the HLR results in one read transaction issued against the HLR. Again, using assumptions from [72], each subscriber makes 1.4 calls per hour. This gives a load of almost 3900 read transactions per second. It is commonly required that 95% of calls are set up in 5 to 50 milliseconds [50].

**Trading Systems** Trading is the short-term selling or buying of stocks and other equities, with the intent of making money by utilizing short-term fluctuations in the market. Generally, the positions in the securities are shorter than one week. Security trading systems are designed to find good trading opportunities by monitoring the market. The system must discover such opportunities and execute the trade before they vanish [33].

The inputs to a trading system include the ticker (trades made) and orders placed for each security, prices of raw materials, and indices. For instance, if a trader follows oil-related companies, the input could be the tickers from all oil companies at all stock exchanges worldwide, the crude oil prices in USA, Europe, and Asia and indices related to energy and oil across the planet. The processing of such inputs should be made as fast as possible to enable the system to react quickly. A trade can be won or lost in milliseconds. A quicker response could mean more income, which means the processing should be as real-time as possible. In addition, the more information a system can handle, the more reliable trades are achieved.

The possible amount of input is huge. For instance, the New York Stock Exchange has up to 14 million trades in one day. For each trade, both a seller and a buyer place a bid, and the trade is executed, causing at least two updates to the trading system. Assuming that 25% of the transactions happen at the peak hour, the system must be able to receive around 2.000 updates per second. To watch other securities and markets, and handle the top short-term peaks of trading, the system must be able to handle higher throughputs as well.

Any unavailability could be financially devastating. While the system is down, the rest of the market would continue trading, causing many lost trading opportunities and, even worse, massive losses could occur since the system is not able to sell stocks, which then plummets. Thus, a trading system should have high availability.

The discussion above shows that trading systems require high availability and real-time responses and, thus, are suitable as a motivating example for this thesis.

**Online Gaming** The gaming industry has evolved from an annual sale of USD 2.6 billion in 1996 to USD 7.4 billion in 2006 [5]. As more people get Internet access, the number of online games and gamers grow as well. In 2006, forty-four percent of most frequent gamers play games online. A special type of online games is called *Massively Multiplayer Online Games* or MMOG for short. These games enable the user to interact with many other users in a virtual game world. In 1991, the limit was around 50 simultaneous players. In 2000, there were many with thousands of players, and the record to date is 41,690 simultaneous players in one game [27]. There were at least 13 million subscriptions of MMOGs in 2006 (exponentially growing), and the worlds largest MMOG subscriber base was the 7 million subscribers of World of Warcraft [70]. In January 2008, the number of monthly subscribers passed 10 millions [110], however, they are currently partitioned to separate “universes” of a maximum of around 4.000 players.

Guild Wars is a MMOG, which can handle 2,500 – 3,500 players per game server. The database (Windows SQL server – Enterprise Edition) handles the load spikes of up to 5,000 transactions per second. However, as noted, the database is a bottleneck, and a scale-out is needed to add more features to the game [73].

Typically, MMOGs require high throughput (tens of thousands of simultaneous users produce a lot of input and need a lot of output), high availability (gamers frequently threaten to quit playing when servers are unavailable) and real-time responses (the gamer must be presented the new state of the game in milliseconds, for instance, to avoid being killed in combat). Thus, main-memory databases and the work in this thesis should be considered.

**Sensor Networks** A sensor network consists of a number of sensors (both wired and wireless) which report on the status of some real-world condition. The conditions include sounds, motions, temperature, pressure and moisture, velocity, etc. The sensors send their data to a central system that makes decisions based both on current and past inputs. To enable the networks to make better decisions, both the number of sensors and the frequency of updates should be increased. Thus, sensor networks must be able to tolerate an increasing load. For applications such as health care in a hospital, automatic car driving systems, space shuttle control, etc., data is needed in real-time, and must be extremely reliable and available, as any unavailability

or extra delay could result in deaths.

**Roaming Between Different Networks** IPv6 extend the number of IP addresses to enable all devices to have a separate IP address. Such devices may be mobile and can move away from the home network. Mobile IPv6 is an internet protocol to manage this movement. When a device moves to another network, then the IP address of the device is changed. To keep the existing connections open, the old IP address must be linked to the new one for each service it is currently using. This must be done in a *secure* and *reliable* way. In today's system, a handover takes 2 seconds. For some applications, this delay is unacceptably long. To reduce this time, a system with less delay is needed.

### 1.1.1 The Challenges of Main-Memory Commit Processing

At the end of transaction execution, a termination protocol is executed to ensure ACID properties [44]. This *commit processing* has been found to take up to one third of the total processing time [101]. For distributed systems with network messages and delays, the relative cost has been found to be even higher [59, 45]. For this reason, commit processing is a widely studied subject. The research community has focused on both increasing the resilience and improving the performance of these ever since the original paper on the two-phase commit protocol by Jim Gray [37] was published 30 years ago.

However, research of commit processing in replicated main-memory systems has been lacking. The classical challenges of distributed databases apply to main memory as well: How to improve the performance during normal failure-free operation and at the same time increase the resilience to failures. Both challenges affect the other one. Here, they are presented as two separate entities, and special issues related to main memory are noted.

#### Performance during Failure-Free Operation

During normal operation, failures are rare. Thus, the most common case is a transaction that encounters no failures and commits successfully and the system should be optimized accordingly. Commit processing should add as little overhead to transaction processing as possible, both as total workload added to the system and latency for each transaction. Main-memory commit processing has a huge potential advantage since it avoids the costly synchronous disk-writes [33]. However, to keep the data and logs persistent,

one or more backups must exist. Updating these backups as a part of the commit protocol requires both communication delays when messages are sent over the network and processing time. Thus, commitment protocols should keep the overhead of updating the backups at a minimum.

### Resilience to Failures

Even though it is a rare event, processes do fail at times. Therefore, the commit processing algorithms must be able to handle such failures. For pure main-memory systems, a crash failure of a process will lead to the loss of all data and logs at the crashed process. Thus, it is important that another process is ready to take over the processing. In addition, to maintain consistency, the system must maintain a backup of the data of the crashed process.

Some systems exhibit *non-deterministic* behavior; i.e., a re-execution of the same request with the same input values does not necessarily result in the same output value (see Section 2.3.3 for more details). Non-determinism can be introduced by timeouts and multithreading, but others also exist. If care is not taken, non-determinism in a server can jeopardize consistency. Assuming that servers can act as clients towards other servers and a request from a non-deterministic client is sent to a server. Then, the client fails, and since there is no guarantee the client sends out the same request again, the request is now invalid. Therefore, its effects should be removed to keep the system consistent. In replicated systems, this matter is made worse, since a backup replica of the client would take over the processing, and this change is *transparent* to the rest of the system. Hence, non-determinism must be managed to ensure consistency.

## 1.2 Research Questions

Based on the discussion in the previous section, the main research question of the thesis is:

**Q-1** *How can commit processing be tailored to main-memory primary-backup transactional systems to improve performance and fault tolerance?*

The special properties of main memory (fast, volatile and limited size) call for specialized commit protocols to exploit the performance potential. Performance is both system throughput and transaction response time. This means that both the total amount of work done for each transaction and the latency of each transaction should be minimized.

The possible inconsistencies caused by replicated invocations in a non-deterministic environment should be managed. This must be done while keeping in mind the requirements of Question Q-1. Thus, an additional research question is needed:

**Q-2** *How can non-determinism be managed in main-memory primary-backup transactional systems, while adding minimal overhead?*

Transactions are designed to keep a system consistent, as a part of the ACID properties. Therefore, they seem well suited to handle the possible inconsistencies arising from non-deterministic execution. However, since the group communication (replication) system is designed to hide the replication from any application, replica failures are also hidden from the transaction manager. Thus, this issue must be handled in transactional systems as well, while adding as little overhead as possible. Since failures are rare, extra overhead during failure-free scenarios should not be traded for reduced overhead after failures.

### 1.3 Research Methodology

Research in the field of computer science has been divided into three categories [19]: Theory, abstraction and design. This thesis fits into the *Design* paradigm. It is characterized by its engineering roots, and its goal is to construct a system that solves a *problem*. Its phases are:

1. State requirements
2. State specifications
3. Design and implement the system
4. Test the system

The research presented here aims at solving the *problems* of improving the performance of main-memory commit processing and managing non-determinism.

To be able to *state requirements*, the solutions must be applicable to existing transactional database systems. Therefore, common terminology in the field of distributed systems and related research must be understood. Then, we *state specifications* for the methods to solve the problems in Section 1.1.1. To verify the validity of the methods, they are *designed and implemented*, and finally the system is *tested*.

Evaluation of the various commit processing protocols and techniques can be done by real experiments or simulations. *Real experiments* are important as they allow testing the methods on a computer that is more complex than the model used to design them. However, the required hardware may not be available and experiments across many processes are hard to reproduce. Actually, it is impossible to guarantee that processes in a cluster all starts in the exact same state for two tests. Therefore, any rigorous comparison between the methods cannot be performed. In contrast, *simulations* can be performed with limited hardware and the experiments can be fully controlled. The initial state of the system can be guaranteed to be the same and two methods can be compared. Real experiments have been used in Paper 1, while Papers 2 – 6 use simulations.

Simulation results should be *verified by statistical analysis*. In this thesis, it is based on queuing theory, and it is important to validate the *sanity* of the results. Statistical analysis has been used in Papers 2 – 6.

## 1.4 Thesis Organization

This thesis is divided into two parts. The first discusses the background and context for the research. Chapter 1 presents the motivation and challenges of the topic together with the research questions and approach. In Chapter 2, fundamentals of fault tolerance in distributed systems such as terminology, replication, and transaction are presented. Chapter 3 is a survey of existing techniques, protocols, and optimizations for commit processing and solving the orphan request problem. The first part is concluded in Chapter 4 by summarizing the main contributions of this research, discussing some limitations and proposing directions for future work.

The contributions of this thesis are gathered in the second part. Paper 1 addresses the issue of handling orphan requests in a non-deterministic transactional system. Paper 2 presents commit protocols for main memory primary-backup systems. Paper 3 presents dynamic coordinator optimizations for main-memory commit processing. Paper 4 uses priorities and piggybacking to improve further on these commit protocols. Paper 5 generalizes the protocols for use in multiple backups systems. Finally, Paper 6 examines how to use the type of transaction to improve the performance of systems with many short write-only and read-only transactions.



# Chapter 2

## Fault Tolerance in Distributed Systems

In this chapter, a brief introduction to dependability and general fault-tolerant computing is given. Then, distributed systems and methods for achieving fault tolerance in distributed systems, replication, and transactions are presented.

### 2.1 Dependability and Fault Tolerance

The exact meaning of fault tolerance is dependant on the context. Here, the context and terminology used in this thesis are defined.

#### 2.1.1 Terminology

The **dependability** of a system is how much trust can *justifiably* be placed on a service based on the *quality of the delivered service* [62].

A **service** is a collection of operations that can be invoked by the users of the service or the passage of time [21]. A user is another system or component that can be either hardware or software. A **server** is an entity that implements a service and processes the *service requests* from the users. A server is **correct** if it executes according to an agreed-upon description of the expected service, the **service specification**. If server  $S$  executes *correctly* only if server  $R$  executes correctly then  $S$  **depends** on  $R$  [21].

When there is a deviation between the delivered behavior and the specified behavior of a service, a **failure** has occurred. An **error** is a part of the system state that is *erroneous* and can cause a failure. The cause of an error is a **fault**. Faults can be introduced during all lifecycle phases of a system,

thus, *design faults*, *implementation faults* and *operational faults* can exist. They can also be classified into *transient*, *intermittent* or *permanent* faults [61]. Hence, a *latent* error is created by a fault. The error becomes *effective* when it cause the service to be incorrect. An effective fault is called a *failure* [62].

To achieve a dependable system the combined uses of different types of methods must be used together [62]:

- **Fault-avoidance:** The *prevention* of faults at the time of construction.
- **Fault-tolerance:** Provide a well-defined failure behavior or mask component failures to users [21].
- **Error-removal:** Using *verification* to remove latent errors.
- **Error-forecasting:** The *estimation* of the kinds of errors and their frequency and impact.

A processor can have both *volatile storage* and *stable storage*. **Stable storage** [60] survives a processor crash, while volatile does not. In disk-based systems, the first is typically a disk or tape device, while the latter is main memory or caches. In replicated systems, the main memory from multiple shared-nothing processors can be used to create stable storage.

### 2.1.2 Module Reliability and Availability

The **reliability** of a component is the time from the starting moment to the next failure. It is quantified as *mean-time-to-failure* or MTTF for short. The service outage time is called *mean-time-to-repair* or MTTR for short. The **availability** of the system is the ratio between service time and total time, which is quantified by  $(MTTF/(MTTF + MTTR))$  [39].

The availability of a system has been classified into the number of nines in the percentage of time it is available [39]. Many systems require a downtime of less than five minutes per year. The availability of such a system would have to be 99.999%, or five nines. To facilitate the high availability requirement of the entire system, each of the *relied upon* subsystems or components must have an availability of six nines.

### 2.1.3 Failure Classification

A system can display various kinds of failures. These failures can be classified into four different types of failures [21]:

- **Omission failures:** A server does not respond to an input.
- **Timing failures:** A server responds correctly, but it arrives too early or too late. When it arrives too late, it is called a *performance failure*.
- **Response failures:** A server responds incorrectly. Two types of response failures exist: A *value* failure occurs when the output is incorrect, and a *state transition* failure occurs when the internal state transition of the server is incorrect.
- **Crash failures:** Following an omission failure, the server does not respond to inputs until it restarts. An *amnesia crash* has occurred if the server restarts in the initial state. If some of the state from before the crash is retained after restart, a *partial-amnesia crash* failure has happened. A *pause crash* failure has occurred if all state from before the crash is retained. A *halting crash* occurs if the server is never restarted after a crash.

A component is *failfast* if it stops working after a fail has been detected [39]. This can be beneficial since transient bugs are removed from the system, while transactions make sure the system stay consistent. Processes that act in this way are also called fail-crash processes.

An execution is *failure-free* if no failures happen during the time span that is currently studied. The time span can be the time to process a request or a transaction.

#### 2.1.4 Failure Semantics

To a programmer of recovery actions it is very useful to know which kinds of failures the server is likely to display. Otherwise, every possible failure scenario would have to be handled. Thus, it is important to specify which types of failures (omission, timing, response, crash) a server is likely to exhibit. This is called the **failure semantics** of a component [21].

For example, if a component has a tendency to respond too late, the component has *performance failure semantics*. If a component can display *any* failure, the component has *arbitrary* failure semantics [21].

There will always be a chance that arbitrary failures can occur. Thus, failures not contained in the failure semantics of a component can occur. They are called **catastrophic failures** [21] and the probability of such failures should be specified.

### 2.1.5 Failure Masking

Well-defined failure semantics for a component is important. It provides a way for the system to mask failures of the component. A failure is **masked** by either hiding all its consequences or changing it into a more benign type of failure. Two types of failure masking exist: Hierarchical failure masking and group failure masking. These two can be used separately or together. First, hierarchical failure masking is presented and, second, group failure masking is introduced.

#### Hierarchical failure masking

A user of a server can mask the server's failures. This is called **hierarchical failure masking** [21]. By looking at the failure semantics of a server, the programmer of another server that *depends on* it can mask the failures for its users. Consider a server  $S$  depending on a server  $R$ .  $R$  has omission failure semantics. If  $S$  notices that  $R$  did not respond to some input, it can resend the input until  $R$  responds. This masks the omission failure of  $R$  for a user of  $S$ , but can cause other failures. For example, if response time requirements of  $S$  is stringent,  $S$  may exhibit performance failure semantics because of the omission failure of  $R$ .

#### Group failure masking

A **group** is a collection of servers executing the same service. A server in a group is called a **member**. It can logically be seen as one unit from the rest of the system. A server failure can be masked by such grouping. **Group failure masking** logically groups independent, physically distributed servers together to provide a correct service despite failures in some of them [21]. The group will produce a *group output* and the individual server's failures are masked. Depending on the implementation and requirements, the output can be a majority vote for the member outputs or the output from a fastest member or a special member of the group. For any group, the maximum number of members that can fail while the group still operates correctly is  $k$ . If  $k$  is one, the group is *single-fault* tolerant, and if  $k$  is larger than one, the group is *multiple-fault* tolerant [21].

### 2.1.6 Liveness and Safety

Both liveness and safety are properties needed to make a system available and consistent. The first property causes a system to eventually do something good (i.e., the eventual execution of input and the delivery of output

according to the service specification), while the latter causes a system to do nothing wrong (i.e., not leaving it in an inconsistent state) [4].

Systems can implement both liveness and safety mechanisms. Liveness is a *forward* recovery concept: The system tries to construct an error-free state from the current, erroneous state. Section 2.3 presents a liveness mechanism. Safety on the other hand is a *backward* recovery concept: The system removes the erroneous state by undoing actions until a correct state is achieved. Section 2.4 introduces a safety mechanism.

## 2.2 Distributed Systems

A **distributed system** is composed of independent physical *nodes* (interchangeably *processes*) connected through a network. A node is a computer comprised of software and hardware components. All communication between the nodes is performed by sending messages over the network. The distribution of the nodes is *transparent* to a user of the system, i.e., it is not possible to see the distributed nature of the system for a client.

The next sections present the characteristics of distributed systems and distributed applications.

### 2.2.1 Characteristics of Distributed Systems

Message delivery can be reliable or unreliable. A **reliable** communication channel has these properties:

- It does not lose messages; i.e., any message sent from one process is eventually delivered to another process.
- It does not change messages; i.e., the message delivered is identical to the sent one.
- It does not copy messages; i.e., no message is delivered twice.

Messages can be grouped together to save processing and communication overhead. This is called **piggybacking** [11] and is used to increase the throughput of a system at the cost of delaying some messages until there are more going to the same destination or the wait is determined to be too long.

### 2.2.2 Distributed Applications

A **distributed application** is a program executing on *virtual nodes* in a distributed system. The virtual nodes are spread out across the physical nodes

(computers). They work together to supply a service by sending messages through the network. Many different applications may exist in the same distributed system. Each virtual node (server) offers a service to the others. A user of the server is called a **client**. From now on in this thesis, a **server** is a process executing a virtual node program. However, the concept of a server is not static. A server can act as a client to another server and so on. Thus, a single virtual node can be both a server and a client at the same time, but for two separate requests.

### 2.2.3 Real-Time Systems

A real-time system must be able to perform (some of) its tasks within a given deadline. For instance, there may be a maximum time from a real-world event to the system reacts. In a *hard real-time* system, completing a request after its deadline is useless and may lead to the failure of the system. However, in a *soft real-time* system, some late requests are tolerated, but too many of them lead to decreased quality of service.

For both *real-time* and *non-real-time* systems, the response time of a transaction can be measured. In this thesis, the response time of a transaction is defined as the time from the transaction request is sent from the client until a reply has been received. The length of the response time depends on the total *delay* of the transaction. The delay is determined by the longest path of operations that has to be completed in a serial manner. This is also called the *critical path*. To reduce the response time of a transaction, either the delay or the total load on the system (which would make queues shorter) must be reduced.

## 2.3 Replication in Distributed Systems

Replication is a technique used to achieve three improvements for distributed systems: Performance enhancement, increased availability, and fault-tolerance. These can be accomplished all at once.

In a distributed environment, any process may fail at any given time. A **single point of failure** is a component that, if it fails, causes full or partial loss of the provided service of a system. For a user, the unavailability can be perceived as anything from a slight nuisance to highly critical, depending on the type of service and the length of the outage. Examples of components include routers, communication lines, CPUs, disks, etc.

Group failure masking (see Section 2.1.5) solves the problem of single points of failures by introducing redundancy; having more than one copy

of each component. The copies are kept together in a logical group, and a member of the group is a **replica**. This enables the system to stay operational, despite failures. In the domain of distributed systems, redundancy, amongst others, can be introduced by executing the same service on multiple processors. This is called **replication** of processes. This thesis focuses on the replication, thus other types of redundancy are outside the scope.

Replication of servers may seem straightforward, but to ensure both consistency and availability of the system, special care must be taken. For instance, group management is needed to manage each replicated group. Depending on the type of replication or *replication strategy* used, the demands from the group management facilities change.

The rest of this section presents general group communication, replica consistency, and replica determinism, and the most common replication strategies.

### 2.3.1 Group Communication

*Group communication* was first presented in the context of distributed operating systems [18]. The fundamental idea is to be able to treat the entire group as one entity [43]. This requires coordination, i.e., some form of internal communication, within the group to maintain and manage it and to keep the global group state consistent. Group communication must ensure two properties [109]: Agreement properties and ordering properties. The first guarantees that the members agree on certain things, while the latter ensures that all members see the events in the same order.

Many group communication kits are available, some for academic research and some for commercial use: ISIS [12], TOTEM [75] and Transis [26], MAESTRO/ENSAMBLE [107], Phoenix [67], JGroups (formerly JavaGroups) [8], CORBA OGS [30], Eternal [76], Jgroup [69], IRL [7] and JMSGroups [57]. For a thorough presentation, the reader is referred to [7] and [56].

The rest of this section presents the agreement and ordering properties, and touches upon the problems related to process failures.

#### Failure Detectors

The *FLP impossibility result* [31] states that, in an asynchronous system, processes cannot be guaranteed to reach an agreement in the presence of a process crash. The cause is that a crashed process cannot be distinguished from a slow process. However, this result can be circumvented by introducing failure detectors [17]. A **failure detector** is a distributed oracle, which tells the correct processes whether a process is faulty or not.

Failure detectors have been classified according to the mistakes they can exhibit [17]:

- Completeness
  - ◊ Strong Completeness: Eventually every process that crashes is permanently suspected by every correct process
  - ◊ Weak Completeness: Eventually every process that crashes is permanently suspected by some correct process
- Accuracy
  - ◊ Strong Accuracy: No process is suspected before it crashes
  - ◊ Weak Accuracy: Some correct process is never suspected
  - ◊ Eventual Strong Accuracy: There is a time after which correct processes are not suspected by any correct process
  - ◊ Eventual Weak Accuracy: There is a time after which some correct process is never suspected by any correct process

As failure detectors are unreliable, except for the perfect failure detector with strong accuracy and strong completeness, processes are not marked as failed, but as *suspected to have failed* or just *suspected*.

## Agreement

To be able to cooperate the members on a group must agree on certain things: (1) on some value, (2) the set of members and (3) the delivery of a message. The first is the Consensus Problem [83]. The group must choose a value out of a set of one or more values proposed by one or more members of the group. They must agree even though processes may crash in the middle of the consensus protocol. It has been shown that consensus can be solved in an asynchronous system augmented with an unreliable failure detector if the majority of processes are correct [17].

The second, agreement on the set of members, is managed by a *group membership protocol*, GMP. It is responsible for keeping a current **group view** or **view** of the group and to avoid it being a single point of failure, it must be distributed. GMP works like this: If a **Leave** message is received from a member, it is removed from the view. Similarly, a process is included in the view by sending a **Join** message to the service. Any change in the set of members in the view is called a **view change**. If view changes are allowed GMP is called *dynamic*. If not, i.e., the set of members stays the same during the entire lifetime of a system, it is called *static*.

If a group member fails, it will not send out a **Leave** message. Therefore, the *failure detector* is responsible for updating GMP with the suspected processes. These are removed from the group. Messages from processes not currently in the group are not delivered, and processes not in the group do not deliver messages from the group.

A message that is intended for all members of a group must be delivered by all or none. The third, agreement on the delivery of a message, covers this. *Basic multicast* [78] is executed simply by making the sender process send the message to each member of the group one by one. However, if the sender fails after it has sent to some, but not all of the members, they will not *agree* on whether the message was delivered or not.

Using *reliable multicast* [78], delivery of messages is guaranteed to be once and only once for all correct members.

All agreement protocols can be made *uniform* [78], by ensuring that, if any member (failed or not) delivers the message, then all correct members of the group will eventually deliver it.

A communication failure can cause a group to be divided into one or more *partitions*. A **partition** is a subset of the nodes contained in the group. For some applications, each of these partitions can process requests on its own and its state can be merged later when communication is restored. Others, however, can only process requests at the majority partition.

## Ordering

The processes must be able to agree on the ordering of events. An **event** is an action in the system. Events are distributed to other processes through the communication channels using messages. Ordering can be imposed on reliable multicast to fulfill the requirements of the system specification: *First-In First-Out (FIFO)*, *Causal* and *Total Order*. The first ensures that if two messages were sent by the same process, it is delivered by each receiver in the same order.

The second ensures *causal ordering*. This means that, if a message  $m$  was delivered by a process  $p$ , before  $p$  sent out message  $n$ , then all other processes will deliver  $m$  before  $n$ . Causal ordering implies FIFO.

The third is also called *atomic multicast*. It requires that all processes deliver all events in the same order. It has been shown that atomic broadcast is reducible to the consensus problem [17].

### 2.3.2 Replica Consistency

When multiple replicas of data exist, consistency must be handled. This applies to both replication for fault tolerance and replication for performance and availability. The **consistency model** of a group is the behavior its clients can observe. To achieve fault tolerance using replication, the replicas must obey a *strong* consistency criterion, i.e., sequential consistency [58] or linearizability [47]. Both require the following criteria to be satisfied [105]:

- The result of any execution by all processes is the same as if the operations were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

Linearizability is stronger and has the following additional requirement [105]:

- If the timestamp of an operation is smaller than the timestamp of another operation, then the operation with the smaller timestamp should precede the other in the sequence.

Using linearizability, each replica must ensure that it has executed every preceding operation before executing the next one. This requires the system to wait until it is guaranteed that no preceding operations arrive late. Because of this requirement, an asynchronous system cannot ensure linearizability. Thus, asynchronous systems use sequential consistency.

### 2.3.3 Replica Determinism

Some replication techniques, like active replication [98], require all replicas to behave identically. If, for instance, there are two replicas of a server group giving diverging replies to a client, it has no way to know which result is the correct one. In addition, if the internal state changes of the server replicas differ, it leads to inconsistencies that may later spread to other parts of the system. Therefore, all *correct* replicas must agree to the same result. This can be done by making the execution of the replicas *deterministic*.

Schneider [98] defines replica determinism as follows: “A replica group is deterministic if, in the absence of faults, given the same initial state for each replica and the same set of input messages, each replica in the group produces the same ordered set of output messages.” Poledna [86] argues that this definition is too restrictive, as it does not cover all replication techniques and it does not capture the timing requirements of real-time systems. For this thesis, however, the above definition by Schneider suffices.

Many sources cause non-deterministic execution. The most obvious one is a mathematical random function, but more subtle sources are also at work. A timeout is one example: Say, for instance, that a server group waits for a reply from another server and the reply is received close to the expiration of the timeout. Some servers may decide that it came just in time, while others decide it came too late. Even with global coordination of the timeout, their respective decisions can diverge because of tiny differences in the processing speed or network delay. Other sources of non-determinism include inconsistent inputs from analog sensors, multithreading, inconsistent order of requests, and non-deterministic programming-language specific constructs [86].

### 2.3.4 Replication Strategies

There are different ways to do the processing of request within a replicated group. These are called **replication strategies**. They decide how the group is addressed both from an intra group and a inter group perspective. There are two main types of replication: *Active* [97] and *passive* (or the *primary-backup approach* [14]). Other types of replication, e.g. semi-active [108] and semi-passive [23], also exist.

The rest of this section presents these four types of replication. Quorum replication is not discussed here.

#### Active Replication

One way to perform replication is that all members of the group *actively* receive, process, and reply to all requests in parallel. This is called active replication or the state-machine approach [98]. In addition, they are all equal, i.e., there is no distinguished member or leader of the replicas.

Active replication is illustrated in Figure 2.1. A client invokes a service with three replicas by multicasting the request. Each of them processes the request, and replies to the client. Depending on the policy, the client can choose to wait for a reply from all replicas, a majority of them or just the first reply before it continues processing.

Group communication, discussed in Section 2.3.1, is required to ensure that all of the servers see the same set of messages and thus stay consistent. In addition, as mentioned in Section 2.3.3, replicas must be deterministic to avoid jeopardizing consistency.

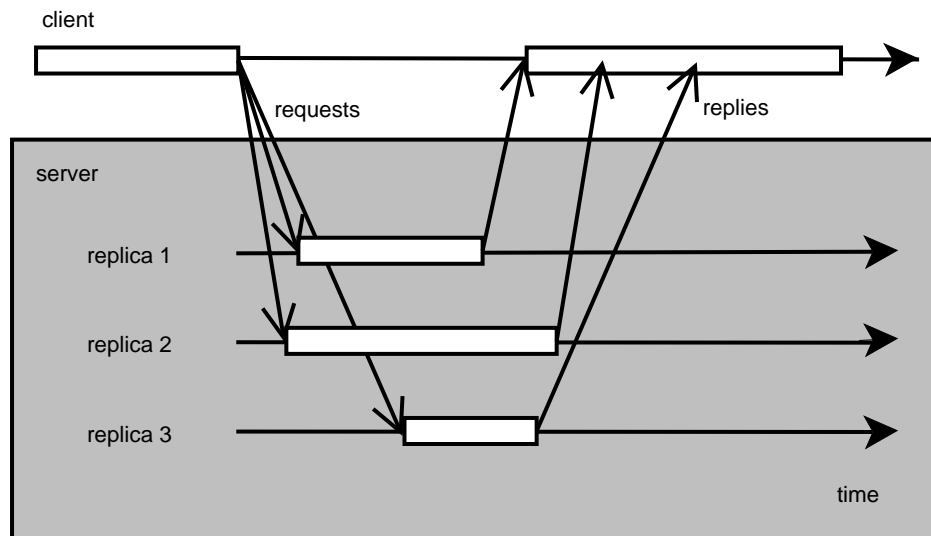


Figure 2.1: Active replication

### Passive Replication

In passive replication, or the primary-backup approach [14], only one replica is active. That replica is the leader (primary replica or just **primary**) of the group, and it receives, processes, and replies to requests from clients. The primary updates one or more backups periodically or at certain events. Since there is only one replica processing the requests, non-determinism can be allowed.

The backups modify their own state according to the updates received from the primary and in the case of a primary failure, the backups elect a new primary to take over the processing. This election is carried out by a group management protocol as discussed in Section 2.3.1. These mechanisms are called a *failover* and a request is said *failover* to the backup if it is sent to the backup instead of to the failed primary.

Figure 2.2 illustrates the passive replication scheme. The client sends requests only to the primary, and only the primary processes them. At certain intervals, the backups are updated. In the figure, the backups are updated in a synchronous manner (using acknowledgement messages) after the primary has finished processing and before it replies to the client. Asynchronous updates and periodic checkpointing [11] can also be used.

### Semi-Active Replication

This is a hybrid technique that allows both non-deterministic execution and active replication. One of the replicas is called the **leader** while the others are

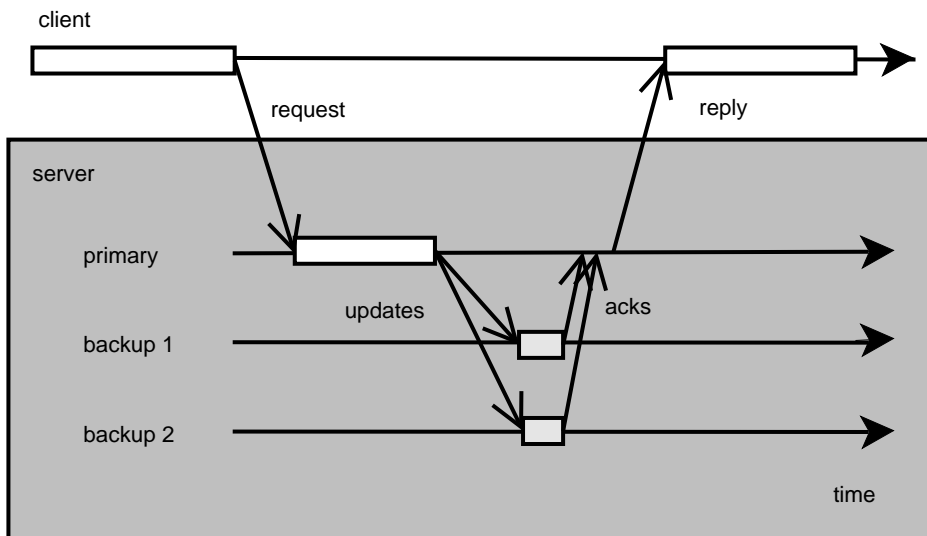


Figure 2.2: Passive replication

called **followers** [108]. As in active replication, all the replicas process the request, but it is up to the leader to process the non-deterministic parts and notify the followers. The leader decides on the ordering of requests and sends its decisions to the followers. In addition, only the leader's reply is received by the client, because identical replies are removed by the communication system.

This technique has been developed in the context of Delta-4 [88]. It is designed to be a dependable real-time architecture. Real-time systems require low response times and need to quickly respond to events. If, for instance, an alarm condition is raised, the event needs to be handled immediately. Thus, the specification also allows pre-emption of execution.

### Semi-Passive Replication

Semi-passive replication [22, 23] is a variation of the passive replication technique. The client-server interaction, the timeout policy, and the selection of the update values differ. As will be shown later, the latter is actually also a selection of the primary. Since only one of the replicas actually processes the request, non-determinism is allowed.

Figure 2.3 shows how semi-passive replication works in the failure-free case. The client starts by sending the request to all replicas. The primary processes the request and proposes an update value to the backups. When a majority of the replicas has agreed upon the value, the primary sends the decision to all backups, updates its own state and replies to the client. As

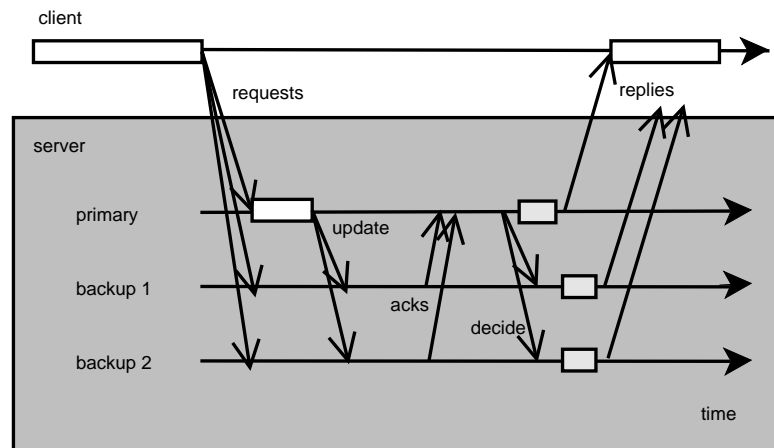


Figure 2.3: Semi-passive replication

each of the backups receives the update, it updates its state and replies to the client. To mask the replication from the client, duplicate replies are discarded by the communication system.

If the primary fails or is suspected to have failed, one of the backups also proposes a value. Since a process in an asynchronous system can be falsely suspected to have failed (see Section 2.3.1), two or more values might be proposed for the same request. A consensus protocol, presented in Section 2.3.1, solves this. This has the effect of masking process failures in the server group from the client.

In standard passive replication, a process is removed from the server group if it is suspected to have failed. When it has recovered, it may join the group again. Both of these operations, which are taken care of by the group membership protocol (see Section 2.3.1), are time consuming and degrade the performance of the system. A timeout mechanism is usually used to detect a failed process. The length of the timeout is a tunable parameter. If an aggressive value is chosen, a slow process might be incorrectly removed from the group, and then subsequently rejoin it. This causes two executions of the costly leader election part of the group membership protocol. If a conservative value is chosen, a failure can lead to a long period where the system is just waiting for the timeout. For time-critical systems, this is unacceptable.

Semi-passive replication avoids this problem by using two different timeout values: an aggressive timeout for the suspicion of a failure and a conservative for the exclusion of a process from the group. After the first timeout, but before the latter, the other processes keep sending messages to the suspected process. As only a majority of the processes needs to answer for the

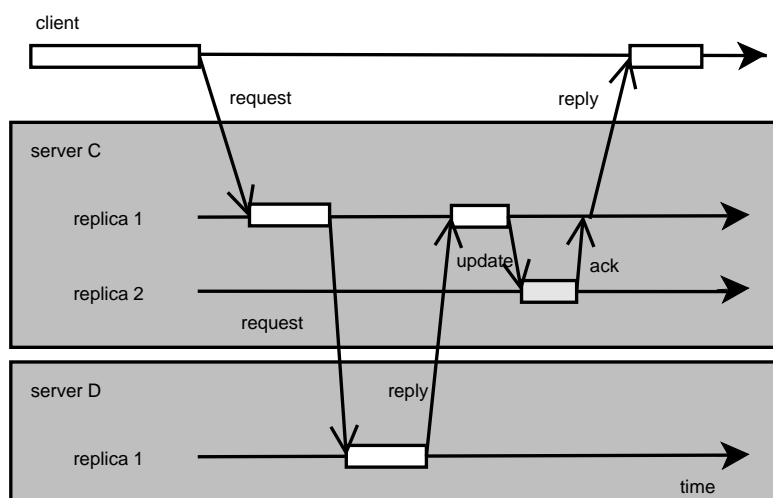


Figure 2.4: A non-deterministic passive server causing a replicated invocation consensus protocol to reach an agreement, this will not affect the performance of the system.

### 2.3.5 Orphan Requests

Passive replication allows non-deterministic execution of replicas. Figure 2.4 shows the failure-free execution of a request by passively replicated, non-deterministic server  $C$ . The primary replica of server  $C$  receives a request from the client and makes a request for server  $D$ . After a reply has been received by  $C$ , it processes the message and updates the state of the backup replica, before replying to the client.

Unfortunately, non-determinism can introduce consistency problems between replicas. If a primary crashes, inconsistencies may spread throughout the system [85, 84]. Figure 2.5 illustrates the situation. If the primary of  $C$  sends a request to  $D$  and subsequently fails, one of the backups becomes the primary. Because of non-deterministic execution of  $C$ , there is no guarantee that the new primary sends an identical request to  $D$ . In fact, it may not invoke  $D$  at all, but rather a different server  $E$  or it may not send a request at all.

Now consider  $D$  processing the request received from the failed, old primary. The processing causes a change in the state of  $D$ . The new state is thus an *inconsistent* state. To make matters even worse, a subsequent invocation to or from  $D$  can leak the inconsistency to the rest of the system. The request from the failed primary is called an *orphan request*. It is crucial that the effects of these requests are undone to avoid inconsistencies.

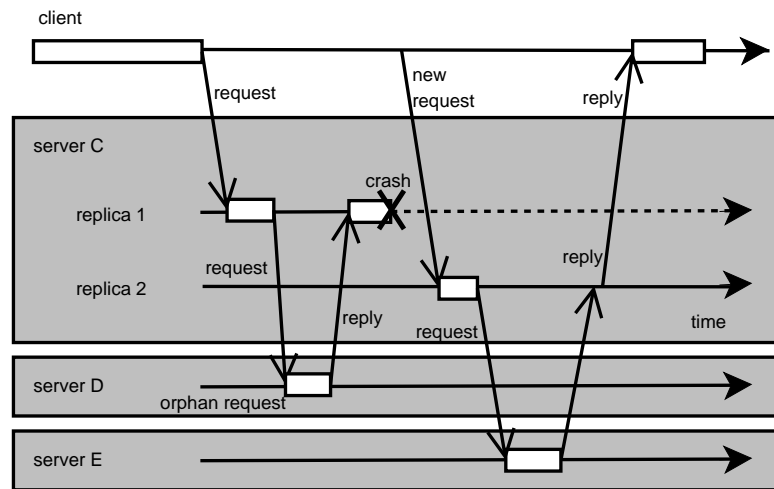


Figure 2.5: Non-deterministic passive replication and a primary failure may lead to an orphan request

## 2.4 Transactions

Transactions were developed in the context of database applications to provide a uniform way to deal with failures. A transaction groups operations together to form an atomic unit to protect user programs from hardware and software failures. In the database domain, a transaction begins with a `Begin` command and continues with a series of `Read` and `Write` operations. More generally, the operations can be anything from a single read or write operation to a complex database query or update. The operations within a transaction can be ordered, thus, they generally cannot execute in an arbitrary order.

Each transaction has an *outcome*. The **outcome** is a decision to accept or reject the transaction as a whole. If it completes successfully, a `COMMIT` decision is reached, or if anything goes wrong, an `ABORT` decision is given. Reasons for a transaction to decide to `ABORT` includes the processing node is out of storage space, a deadlock between two transactions, a local integrity constraint is violated or the user chooses to cancel the transaction.

The rest of this section presents the transaction properties, serializability, concurrency control, recovery and the non-blocking atomic commitment problem.

### 2.4.1 Properties

A transaction is a collection of operations or actions with the ACID properties [38, 44, 11]:

- Atomicity: After the completion of a transaction, all operations of the transaction must have executed successfully or it must appear as none of them was executed. This is also known as the *all-or-nothing* property of transactions or the everyone-or-none property for the distributed systems.
- Consistency: The sum of the operations in a transaction must transform the applications from one consistent state to another consistent state.
- Isolation: The effects of a transaction must not be visible to other transactions before it has committed.
- Durability: The effects of a successfully completed transaction must be permanent once the transaction has committed.

If a failure occurs while a transaction is executing, the transaction is aborted, and all of the state changes caused by the transaction are undone. This rollback of the transaction is the most common implementation of the atomicity property, thus, transactions are a safety-focused mechanism: If a failure occurs, the state is rolled back to a previously consistent state.

The system must ensure the above properties for the *interleaved* executions of transactions and in the presence of failures. Two different set of protocols are used to guarantee this [11, 39]: *Concurrency protocols* and *recovery protocols*. These are outlined below.

### 2.4.2 Concurrency Control

The isolation property must be ensured, also when two or more transactions execute in parallel. In a multi-user system, it is very likely that user transaction will be executed concurrently. Uncontrolled execution of these will probably cause problems like *dirty reads* and *lost updates* [39]. It is the job of the concurrency control protocol [10] to manage the correctness of multiple parallel transactions.

#### Serializability

The simplest concurrency control protocol is to execute transaction in a *serial* fashion: One transaction is started and the next is delayed until the first has completed. This approach, however, results in poor performance since all other transactions are delayed if the active one is blocked while waiting for I/O operations. Thus, transaction should be executed concurrently in a

*serializable* way, as defined by Bernstein and Goodman [10]: An execution is **serializable** if it is computationally equivalent to a serial execution. That is, if it produces the same output and has the same effect on the database (system) as a serial execution. This is called **serializability** [11].

There exist two types of serializability: View serializability [81] and conflict serializability [11]. However, for the purpose of this thesis, it suffices to define *serializability* as the basic idea that if every transaction by itself is correct, then the serial (or serializable) execution of multiple concurrent transactions is correct.

### Concurrency Control Protocols

Protocols for enforcing concurrency control can be classified into two classes: Optimistic protocols and pessimistic protocols. The first executes the operations of each transaction, without checking for concurrency problems until the transaction is ready to commit. When the transaction is ready to commit, it is *validated*. The validation checks if the transaction is serializable. If it is, it is *committed*. If not, the transaction, and possibly others, is *aborted*. This is a potential problem, since a lot of work may be lost due to serializability problems and the performance may go down. Techniques such as locking, serialization graph testing and timestamp ordering all have optimistic versions [11].

The latter, the pessimistic protocols, checks that each operation is serializable before the transaction is executed. Thus, at commit, the transaction is guaranteed to be serializable; there is no need for validation. The most commonly used pessimistic protocol is *Two-Phase Locking* (2PL) [11]. Before it can read or write to any data, it must obtain the lock for that particular piece of data. If any other transaction holds a conflicting lock, it is *blocked* and must wait until the lock is released. Once a transaction releases a lock, no new locks can be set for that transaction. 2PL can result in *deadlocks*, since two transactions may end up waiting for the other to release a lock. Locking can lead to long term blocking, especially for long transactions and during failures. Thus, other variants than locking have been developed (e.g. timestamp ordering [11]).

#### 2.4.3 Recovery

The atomicity and durability transaction properties are ensured by *recovery protocols*. Overall, they ensure that the system always reflects the changes made by committed transactions, and never of aborted ones. This must be guaranteed even in the presence of failures. The unit of failure can be a

transaction, a system and media [11, 39]. A **transaction failure** is a failure causing the offending transaction to abort. The abort can be initiated by the transaction itself or the system.

A **system failure** is a failure causing loss of volatile storage such as the main memory, or just *memory*. The contents of the stable storage, such as the secondary memory (called *disk* from now on) are intact. In distributed systems, **partial system failures** are possible. Those cause the memory of just one or some of the processes to be lost. A typical cause for system failures are loss of power.

When parts of or the entire stable storage is lost, it is called a **media failure**. The cause of this can be disk head crashes. Whereas transaction failures and system failures are non-catastrophic failures, media failures can be a catastrophic failure [34]. A **catastrophic failure** is the loss of parts of or the entire database. For main-memory systems, a system failure is a catastrophic failure as well, and partial system failures can be catastrophic if all replicas of some piece of data fail at the same time.

A set of protocols ensures the system is able to recover to a consistent state after a failure. *Logging protocols* make sure the updates on the state of the system are saved to stable storage (disk or another process) in a consistent way during failure-free execution. After a failure, *recovery protocols* use the logged information to rebuild the system to a consistent state.

## Recoverability

For the system to be able to recover from transaction failures, the execution of transactions must be *recoverable* [11]. To guarantee a **recoverable** execution all transaction that writes some data must have committed before any transaction reading it commits. Thus, if a transaction aborts, all transactions that has read data, which the aborting transaction has written, must also abort.

If one transaction cause another to abort, it is called **cascading abort**. Cascading aborts can hamper the performance by potentially aborting many transactions causing a lot of work to be removed. Therefore, cascading aborts should be avoided. Executions that guarantee to *avoid cascading aborts* are constructed by requiring a transaction never to read data that an uncommitted transaction has written [11]. When it tries to read, it is blocked and must wait until the other transaction has committed or aborted and changed the data back to the previous value.

So far, a transaction is always allowed to write or modify data. However, this can cause problems if a transaction decides to abort after multiple transactions have written to the same data. Simply restoring the data to the

previous value can cause inconsistencies. Executions are **strict** if transactions can only read or modify data written by committed transactions [11].

### Logging – Redo and Undo

To be able to handle system and media failures, information regarding the transactions' state must be stored in a safe manner at all time [11]. This information is stored in one or more *logs*. In a typical transactional system, two logs may be necessary: redo log and undo log. The **redo log** contains the information necessary to *redo* the operations done by committed transactions before a failure. The **undo log** contains the information necessary to *undo* the operations done by uncommitted transactions before a failure.

Often, to ensure the durability or *persistency* of log records at certain points of transaction processing, the log records are *flushed* to disk. A flush ensures that the log record is *physically* stored on disk and not only in a buffer. Meanwhile, the flushing process cannot continue its execution. This is also called *forcing* a record to disk, or *force-write*.

### 2.4.4 Transactional Systems Architectures

For transactional systems, the hardware architecture can be classified into three strategies [39]:

**Shared nothing:** Each processor has its own dedicated memory. All accesses to the data in that memory is coordinated by the processor. To access data at remote processors, the processors exchange messages through a communication network.

**Shared disk:** Each processor has *some* private memory, but there is also data on disks which are *globally* accessible by all processors.

**Shared memory:** This is typically a multiprocessor system. All processes have access to all data. The underlying hardware manages the accesses such that multiple processors can operate on the data concurrently.

The first of these, the shared-nothing approach, corresponds to a distributed system, as defined here, and is the main topic of this thesis.

### 2.4.5 Distributed Transaction Entities

A **distributed transaction** is a transaction that involves processing at more than one node. One of the nodes is then selected as the **coordinator**, or it

can be a dedicated node for all transactions. The other nodes, which execute *transactional operations*, are called **participants** or **cohorts**. A **transactional operation** is an operation that is executed as part of a transaction. A participant can act as a coordinator for other participants. It is then called a **subordinate**. The parts of the transactions, which are handled by the participants and subordinates, are called **subtransactions**. For a subtransaction to commit, the entire transaction must commit.

### 2.4.6 Atomic Commitment

A distributed transaction must have *global coordination*. The all-or-nothing (i.e., atomicity) property must be ensured globally as well as locally. Thus, the servers must *agree* on the outcome of the transaction. An **atomic commitment protocol** is initiated by a coordinator of the transaction. If one of the participants cannot commit the transaction, i.e., cannot guarantee its local ACID properties, all cohorts are forced to abort it. This is the *atomic commitment problem*, or AC. However, AC cannot guarantee that the cohorts will not block. Thus, the *non-blocking atomic commitment problem* NB-AC is used instead. The formal specification of NB-AC has these properties [11, 42]:

**NB-AC1:** *<uniform agreement>* All processes that decide reach the same decision.

**NB-AC2:** *<integrity>* A process cannot reverse its decision after it has reached one.

**NB-AC3:** *<uniform validity>* COMMIT can only be reached if *all* processes voted YES.

**NB-AC4:** *<non-triviality>* If there are no failures and no processes voted No then the decision will be to COMMIT.

**NB-AC5:** *<termination>* Every correct process eventually decides.

Unfortunately, NB-AC cannot be solved in an asynchronous system, even if augmented with failure detectors [42]. However, the *non-triviality* condition is too strong since an unreliable failure detector may falsely *suspect* processes to have failed. Thus, the non-triviality condition is relaxed to weak non-triviality:

**NB-WAC4:** *<weak non-triviality>* If no processes are suspected to have failed and all processes voted YES, then the decision will be to COMMIT.

The *Non-Blocking Weak Atomic Commitment* problem is reducible to consensus and is thus solvable in a system with unreliable failure detectors [42].

# Chapter 3

## Survey of Commit Processing

This chapter presents works closely related to this thesis. The topics include general and main-memory commit processing and management of non-deterministic execution in replicated transactional systems.

Many atomic commitment protocols have been proposed during the last three decades. They have been concerned with two issues: Increasing the performance or the resilience of commit processing. Few approaches cover both of these issues. Blocking approaches are presented first and then approaches avoiding blocking are presented. Finally, existing approaches handling the integration of transaction and replication in non-deterministic environments are given.

### 3.1 Commit Processing: Blocking Approaches

During 2PC, if the coordinator and a participant fail at the same time, other participants may be in doubt of the outcome of the transaction, thus unable to release its locks. This will *block* other transaction trying to acquire the same locks. Thus, 2PC-based protocols are *blocking* [39, 11]. The following sections presents such blocking approaches.

#### 3.1.1 Two-Phase Commit

The two-phase commit protocol is the straightforward application of a wedding ceremony to the atomic commitment problem. The transaction coordinator (minister) asks each of the cohorts (the bride and groom) whether they agree to perform the ceremony or not. If they both answer yes, the coordinator tells them to commit (kiss). If at least one of them answers no, the ceremony is not completed. Because of this and its similarity with contract

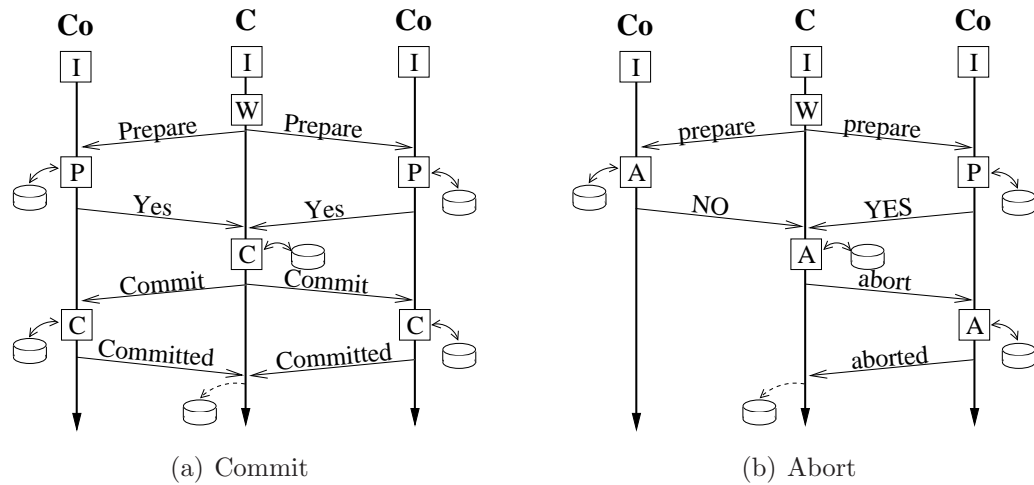


Figure 3.1: The two-phase commit protocol

law, the origin of the two-phase commit protocol cannot be traced back to any one source, but Gray [37] was the first to formalize it in the context of transaction processing.

As the name indicates, 2PC uses two phases as shown in Figure 3.1. The *voting phase* starts when the coordinator sends a prepare message to each of the cohorts. Each cohort then persistently saves its vote (i.e., flushing a *prepare* log record it to disk) before sending it to the coordinator. After all votes have been received, the coordinator decides to commit if all cohorts are ready to commit. The decision is then persistently saved to disk.

During the *decision phase*, the cohorts are notified of the decision made by the coordinator at the end of the voting phase. Each cohort persistently saves the decision. If the decision is ABORT, the changes made by the transaction are rolled back. Otherwise, i.e., the decision is COMMIT; the changes are made durable. Each cohort then acknowledges the decision by sending an Ack message to the coordinator and forgets about the transaction. After acknowledgements have been received from all cohorts, the coordinator saves the outcome to the log and forgets about the transaction.

### 3.1.2 Presumed Abort

The presumed abort (PA) protocol was developed from the R\* distributed database project [71]. It is a modification of 2PC which assumes that a non-existent log record means that the transaction has aborted. In the case of a transaction abort, the cohorts do not need to flush the *decision* log record to disk. This is shown in Figure 3.2. Thus, the performance for aborted

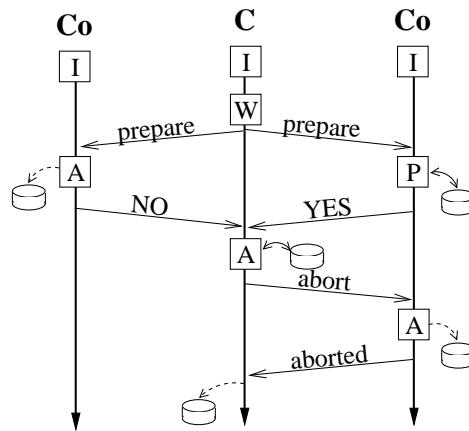


Figure 3.2: The two-phase presumed abort protocol, abort case

transactions is increased compared to 2PC.

PA is the most widespread atomic commitment protocol and has been implemented in many commercial products: DEC's VAX/VMS, Tandem's TMF, Transarc's Encina Product Suite and Unix System Laboratories' TUXEDO. It is also incorporated in distributed transaction processing standards as X/Open and ISO-OSI.

### 3.1.3 Presumed Commit

Whereas PA improves the response time for aborted transactions, the *presumed commit* (PC) protocol [71] tries to improve upon commit processing during failure-free execution. This represents the most common scenario where all goes well and the transaction is successfully committed. Figure 3.3 illustrates the protocol. In the voting phase, the coordinator flushes a *collecting* log record and sends **prepare** to the cohorts. The cohorts reply to the coordinator, which flushes the *commit* log record to disk. During the decision phase, all that is needed is to send the decision to the cohorts, which write the *commit* record to disk at a suitable time. Compared to the 2PC, this causes the same number of flushed log records, but saves a message send.

A further optimization of the *presumed commit* protocol [59] removes the need for the *collecting* log record by retaining some crash-related information; i.e, the range of the identifiers of the transactions that may have initiated but not completed. This is done by keeping a high-water mark and a low-water mark for the identifiers. These must be flushed to disk, but not nearly as often as *collecting* log records. Sometimes it may be necessary to process a transaction with an identifier below the low-water mark. In this case, an *initiating* record must be flushed to disk to be able to recover correctly.

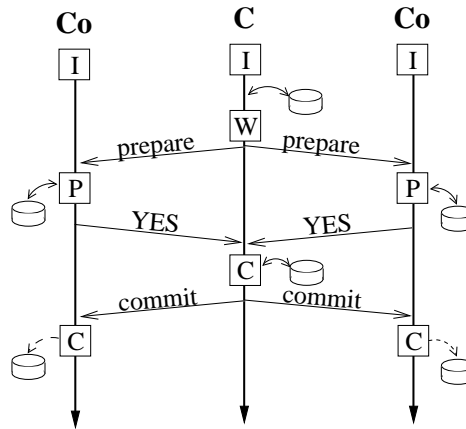


Figure 3.3: The two-phase presumed commit protocol, commit case

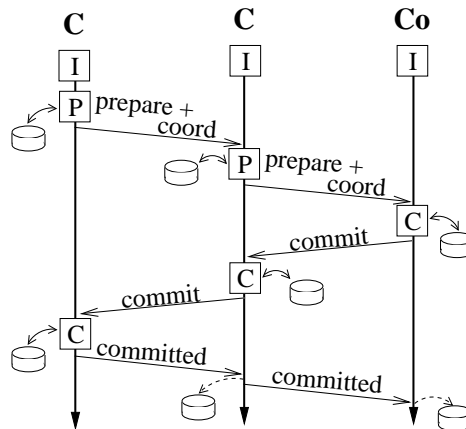


Figure 3.4: The two-phase protocol with linear commit optimization

### 3.1.4 Other Optimizations

Many optimizations have been proposed for commit processing protocols. This section presents the most prominent and important of them.

#### Linear Commit

*Linear commit* [39] is an application of the *transfer of commit* [95] scheme where the authority to make the decision to COMMIT or ABORT can be transferred from one process to another. The cohorts of a transaction are linearly arranged and each of them commits in turn, before the decision is passed back to the original coordinator. Consider Figure 3.4. The original coordinator, process *A*, flushes a *prepare* log record and sends a *prepare* message with a piggybacked *YouAreCoord* message to process *B*. The *YouAreCoord* message

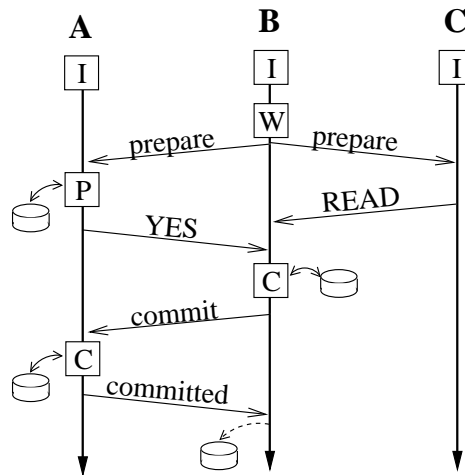


Figure 3.5: The two-phase protocol with read-only optimization

indicates for  $B$  that it is now the coordinator for this transaction. Upon receiving, it prepares itself, flushes a *prepare* log record to the disk and sends a *YouAreCoord* message piggybacked on a new *prepare* message to process  $C$ . Since  $C$  is the last in the chain, it commits the transaction locally by flushing a *commit* log record to disk. The decision is then sent to  $B$  and  $A$  again, both of which flushes a *commit* log record. Then,  $A$  acknowledges the decision to  $B$  and  $B$  acknowledges to  $C$ .

Linear commit has a bad delay but good message cost because parallelism is not used.

### Read-Only Commit

Another general optimization is the *read-only* optimization [71]. A process is allowed to vote *READ* if the transaction being committed has not updated the state of the process. Thus, the process does not need to know the outcome of the transaction. Such a process just releases the transaction's locks before replying and does not need to write any log records. Figure 3.5 illustrates the read-only optimization.  $B$  is the coordinator and sends *prepare* to  $A$  and  $C$ . The transaction has not made any updates from the latter, so it votes *READ*. Thus,  $A$  and  $B$  can complete the transaction without any more interaction with  $C$ .

This optimization can potentially give a very large increase in performance for environments with a large number of read-only transactions.

### Group Commit

Much of the costs associated with commit processing are message sends and flushed disk writes. These can be reduced by grouping them together into a single message or a block write, respectively. This is called *group commit* [35, 25]. Group commit increases the throughput at the cost of delaying some transactions while waiting for other messages or disk writes. An in-depth analysis of this optimization is given by Huang and Li [49] and Spiro, Joshi, and Rengarajan [101]. The extreme case of group commit, *lazy commit*, is where all operations related to commit processing are piggybacked on transactional operations [39].

### Early Answer

Generally, a transaction coordinator does not need to wait until the end of the transaction before it replies to the client of the transaction. Rather it can give an *early answer* when the outcome of the transaction is persistently stored [50]. For disk-based 2PC, the reply can be given once the decision is saved to disk. For main-memory commit processing, the decision must be known by at least two nodes. Hence, the time to execute the decision phase of the commit processing is not included in the response time. Thus, the response time of a transaction, as seen by clients, is reduced at no extra costs.

### 3.1.5 One-phased Commit Protocols

Two-phase protocols add a significant overhead [101, 59, 45] to commit processing, because of two rounds of messages and the flushed writes to stable storage. A suggested approach is to reduce the number of phases to just one to avoid some of the costs associated with commit processing. Several one-phased transaction commitment protocols are presented below.

#### Unsolicited Vote

The idea was introduced by Stonebraker [104] in the context of distributed INGRES. Using this approach labeled *Unsolicited Vote*, UV, each cohort returns YES as part of `Work_Done`. Thus, the voting phase is bypassed and only the decision phase is needed. However, it requires the cohort to prepare the transaction, i.e., check integrity constraints and flush a *prepare* log record to disk, before sending `Work_Done`. If the cohort cannot prepare the transaction, NO is returned.

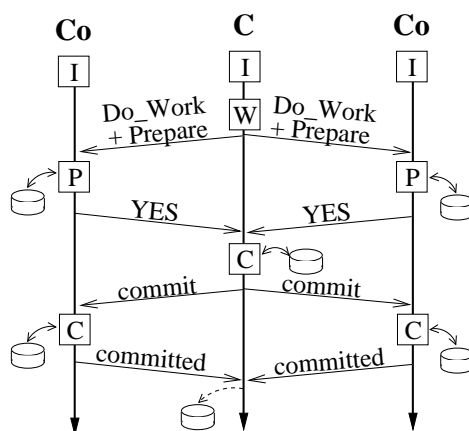


Figure 3.6: The Unsolicited Vote approach

The approach can alternatively be described as piggybacking a **Prepare** on each **Do\_Work**. However, it may cause performance degradation: If one transaction generates multiple, serial **Do\_Works** to a cohort, the cohort must prepare once for each of them, consequently causing multiple flushed writes to the disk. A solution where **Prepare** is only piggybacked if the coordinator knows that it is the last request to the cohort comes to mind. In a setting where a transaction is *precompiled*, this is the case. This way, the unnecessary flushed disk writes can be avoided. This is shown in Figure 3.6.

### Early Prepare

The *Early Prepare*, EP, protocol [102] is a combination of UV and PC. The coordinator is required to flush a *collecting* or *membership* record before sending any **Do\_Works**. If, after a *membership* record has been written, the transaction execution causes the membership to change, i.e., a **Do\_Work** issued to a cohort not already in the membership set, a new *membership* record must be flushed *before* the message is sent. This is illustrated in Figure 3.7. Since *C* is not in the first *membership* record written by *B*, a new must be written before *B* can send **Do\_Work** and **Prepare** to it.

### Coordinator Log

The idea of *sharing the log*, STL, among more than one process was introduced by Mohan et al. [71] and the details were elaborated on by Stamos and Cristian [102, 103]. The *Coordinator Log*, CL, approach is a combination of EP and STL. The entire log for a transaction is put in a single log: The coordinator's log.

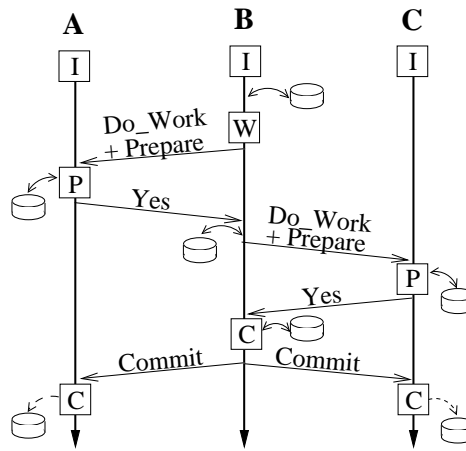


Figure 3.7: The Early Prepare approach. *B* is the coordinator, *A* is a cohort contained in the first *membership* record and *C* is a cohort not contained in the first membership record

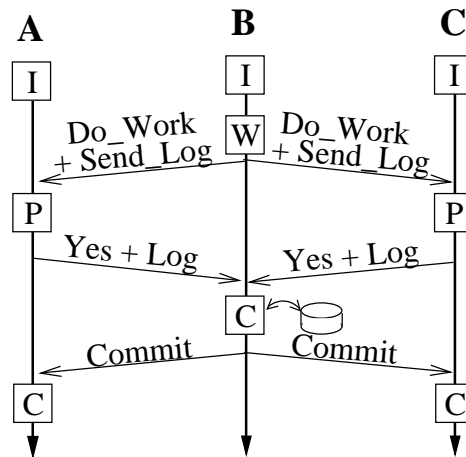


Figure 3.8: The Coordinator Log approach. *B* is the coordinator, *A* and *C* are cohorts.

Figure 3.8 illustrates CL. The coordinator, *B*, issues *Do\_Works* to the cohorts, *A* and *C*, and piggybacks *Send\_Log* messages. Instead of writing to a local disk the cohorts piggyback the generated log records for this transaction on the vote to the coordinator. *B* appends the log records of both *A* and *C* along with a *commit* log record to its own log and flushes it to disk. Then it sends *Commit* to *A* and *C* and forgets about the transaction. The *collecting* record is not needed. In PC, if a coordinator fails after sending out *Prepare*, but before making the decision persistent, it will reply *Commit* to a cohort asking for the outcome. However, other cohorts may have voted NO, and

the reply should therefore have been **Abort**. In CL, however, a recovering coordinator receives log records from all cohorts. Therefore, it does not require the *collecting* log record. Hence, it saves a flushed disk write at the cost of increased recovery costs.

A protocol based on CL for main-memory databases was proposed Lee and Yeom [63]. Because it is main memory, the coordinator only needs to receive redo log records and there is only one disk write per transaction. However, this adds some extra network traffic and a more complex recovery.

### Implicit Yes-Vote

The *Implicit Yes-Vote*, IYV, protocol [3] was designed in the context of high-speed gigabit networks. It is a combination of UV, STL and PA. Each cohort prepares the transaction before replying to the coordinator. However, since the log is sent to the coordinator, the cohort only needs to flush to disk one *start* log record for the transaction even though multiple *Do\_Works* may be received. This is illustrated in Figure 3.9 where cohort *C* received two *Do\_Work*, but only the first one caused a flushed log record. This work assumes Strict 2PL<sup>1</sup>. The protocol achieves one-phased commit processing as the voting phase is executed as a part of the transactional operations. The cost is the same as for PA, except for the extra size of the message because of log shipping.

### 3.1.6 Optimistic Protocols

Optimistic commit protocols are designed to give better response time during normal processing, but will need extra recovery after failures or aborts. They release locks when the transaction is prepared, but must be able to handle cascading aborts by using semantic knowledge [64]. PROMPT [45] uses optimistic locking in the sense that locks can be lent to other transactions after the participant has voted yes. A transaction that lends locks will not reply to the request until the locks are fully released by the previous transaction, and only one transaction at a time can lend a lock. This approach controls cascading aborts to a maximum depth of one. The authors argue the increased concurrency yields better performance.

---

<sup>1</sup>Abdallah, Guerraoui and Pucheral [2] argues that this restraint can be relaxed to SQL2 [51] used by many commercial databases

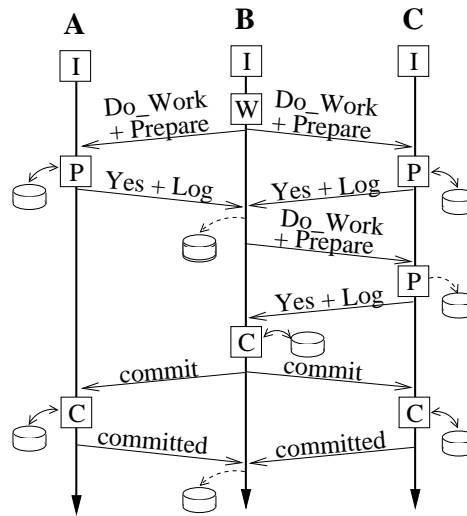


Figure 3.9: The Implicit Yes-Vote approach. *B* is the coordinator, *A* and *C* are cohorts.

## 3.2 Commit Processing: Non-Blocking Approaches

There are three known ways to increase the resilience and achieve non-blocking protocols: Add an (explicit) extra round of messages, add an implicit round of messages (using broadcast or multicast techniques) or replicate the transaction coordinator. These are presented in detail below.

### 3.2.1 Extra Round of Messages

#### The Three-Phase Commit Protocol

The three-phase commit, 3PC, protocol [99] decreases the chance of blocking failures by adding an extra round of messages, thus favoring resilience over performance. 3PC has been extended to partitioned environments [89], and the number of communication steps has been reduced to the same as 2PC by using consensus [41], causing an increase in the number of messages or requiring broadcast capabilities.

In 3PC, an extra phase is added before the transaction can commit. It is called the *pre-commit* phase. This is shown in Figure 3.10. After receiving votes from the participants, *A* and *C*, the coordinator, *B* changes its state to pre-commit, and tells *A* and *C* to do the same. Only after they have acknowledged Pre-Commit is the transaction committed by *B* and the decision propagated.

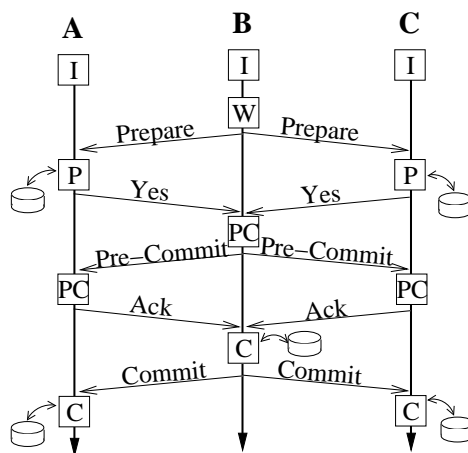


Figure 3.10: The Three-Phase Commit protocol.  $B$  is the coordinator,  $A$  and  $C$  are cohorts.

When aborting, 3PC is identical to 2PC (see Figure 3.1(b)).

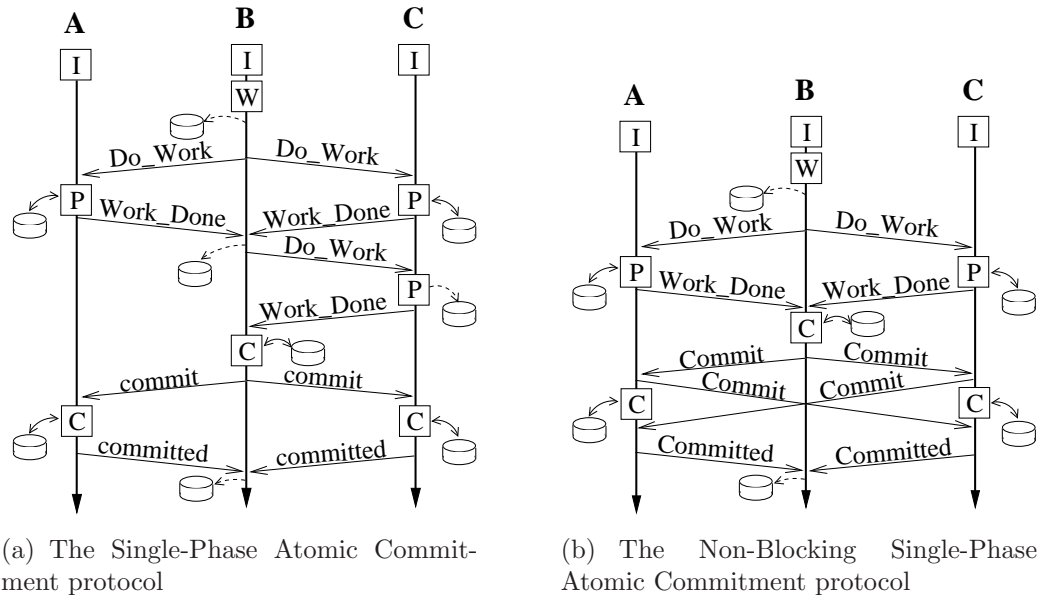
### 3.2.2 Broadcast and Multicast

#### Uniform Timed Reliable Broadcast

Babaoğlu and Toueg [6] proposes a set of broadcast primitives to achieve non-blocking atomic commitment in the context of a *synchronous* communication system. The basic idea is that by reliably broadcasting the decision in the second phase of 2PC, no cohort is ever blocked in an in-doubt state. Each primitive improves upon the previous one. The last is based on *rotating coordinators* [16]. It pessimistically assumes that something has gone wrong before the maximum timeout bound has been reached. The broadcast is reliably carried out by first sending a message containing the decision, the current id of the coordinator and the set of processes to receive the message, to distinguished processes. These will (in turn) take over the broadcasting should the coordinator fail. Then, the message (decision) is sent to all cohorts of the transaction. A protocol, *Uniform Timed Terminating Reliable Broadcast*, where decisions based on timeouts are eliminated was also proposed. The proof for this protocol, however, is not complete.

#### Modular and Decentralized Three-Phase Commit

Two protocols using consensus have been proposed: The *decentralized three phase commit*, D3PC, protocol [40] using *uniform consensus* and the modular and decentralized three phase commit, MD3PC, protocol [41] using *majority*



(a) The Single-Phase Atomic Commitment protocol

(b) The Non-Blocking Single-Phase Atomic Commitment protocol

Figure 3.11: The SPAC and NB-SPAC protocols. *B* is the coordinator, *A* and *C* are cohorts.

*consensus*. The idea for both protocols is that the cohorts send their vote to all other cohorts and the coordinator. Then, all nodes receiving YES from all cohorts distribute the commit decision. In MD3PC, the number of messages is reduced by each cohort only sending the vote to a subset of the cohorts. Thus, some resilience is traded for better performance.

### Non-Blocking Single-Phase Atomic Commitment protocol

The Single-Phase Atomic Commitment, SPAC, protocol and the non-blocking version NB-SPAC was founded on the observation that most DBMSs use pessimistic and “rigorous” concurrency control [1]. Strict 2PL is an example. Therefore, the authors argue that once `Work_Done` is received, then all necessary locks have already been set, and the transaction can be guaranteed to be able to commit unless the cohort fails.

As presented in Figure 3.11(a), the cohort in SPAC flushes to disk a *BeginTransaction* log record before it replies with `Work_Done` to the first `Do_Work` for each transaction. As for IYV in Section 3.1.5, no subsequent `Do_Work` results in a new flushing to disk. However, instead of sending log records from the cohort to the coordinator, the coordinator keeps a log of which operations it has sent to each cohort. This results in site *autonomy*.

NB-SPAC (see Figure 3.11(b)) works in the same way as SPAC except for a change in the broadcasting of the decision. A decision is sent to each of the

cohorts from the coordinator, but this may fail after it has sent to some of the cohorts, but before it has sent to all. This problem is removed by using a reliable broadcast primitive where each of the cohorts, before delivering the decision, sends it to all cohorts. Thus, a situation where only some of the correct cohorts know the decision cannot arise. Failed cohorts are recovered by resending the logged operations from the coordinator.

The underlying assumption of SPAC and NB-SPAC is that the only reason for a cohort to not being able to complete transaction is that it violates the concurrency control mechanism. However, there exist other reasons for a cohort to ABORT. For instance, the cohort can run out of disk space, or the transaction may violate some local integrity constraint. This cause a problem for this approach since a simple replay of the operations may not solve the problems.

#### Optimistic Dedicated Commit Servers

Jiménez-Peris et al. [54] presented a protocol using optimistic dedicated commit servers, here called ODCS. It is based ACP-UTRB presented in Section 3.2.2, but has four substantial differences: First, the vote is multicast to a small group of dedicated commit servers instead of a subset of the cohorts. Second, cohorts that have not yet persistently saved their *prepare* record can piggyback it on the *Vote* in much the same manner as CL in Section 3.1.5. Third, the decision is optimistically delivered by a selected commit server to the cohorts, before the uniform multicast has completed. Finally, when the client receives *Opt-commit* it releases the transaction's locks and set *opt*-locks instead. These locks can be used by other transactions before they are released. These transactions cannot commit until the *opt*-lock it uses is removed. *Opt*-locks are released when the cohort receives *Commit*, indicating that the uniform multicast has been completed at the commit servers.

If something goes wrong for the commit servers after sending *Opt-commit*, causing the uniform multicast not to complete, the commit servers decide ABORT and send the decision to the cohort. They rollback any changes done by the transaction and release the *opt*-locks. Transactions lending the *opt*-locks are also aborted.

#### 3.2.3 Replicated Coordinator

Replicating the transaction coordinator was first done by using process pairs in the context of ENCOMPASS [13]. The idea is that in case of a coordinator failure, the in-doubt cohorts can query the backup about the outcome of the transaction. The number of backups can be varied depending on their

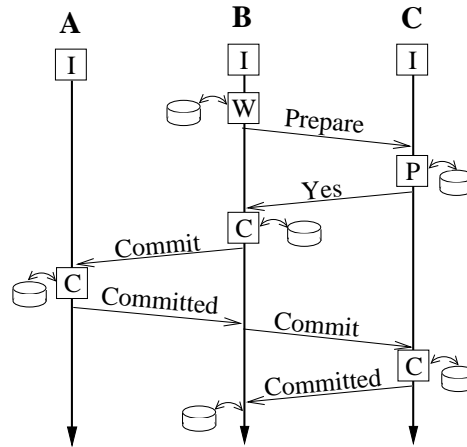


Figure 3.12: The Backup Commit protocol. *B* is the primary coordinator, *A* is the backup coordinator and *C* is the cohort.

reliability. Therefore, as long as not all transaction coordinator replicas fail at the same time, non-blocking is achieved.

### Backup Commit Protocol

Reddy and Kitsuregawa have developed the *backup commit*, BC, protocol [91]. Their approach, which is illustrated in Figure 3.12, has three phases. In the first phase, the primary coordinator logs *start* and sends **Prepare** to the cohort, which logs its decision and replies to the primary coordinator. At the start of the second phase, the primary coordinator logs *DecidedAbort* or *DecidedCommit* depending on the outcome. Then, the decision is then sent to the backup coordinator, which logs the decision and replies to the primary. In the third phase, the coordinator, upon receiving **Ack** from the backup, writes *commit* or *abort* log records and sends the decision to the cohorts. Each of the cohorts logs the decision, acts accordingly, replies to the primary coordinator and forgets about the transaction. After the coordinator has received **Acks** from all cohorts, it logs an *EndOfTransaction* log record.

This approach is not very efficient: The log records are sent to the backup *and* they are forced to disk, causing a decrease in performance. In addition, the backup only finishes transactions already started. No new transactions can be initiated by the backup. This approach has been adapted to multiple backups [90].

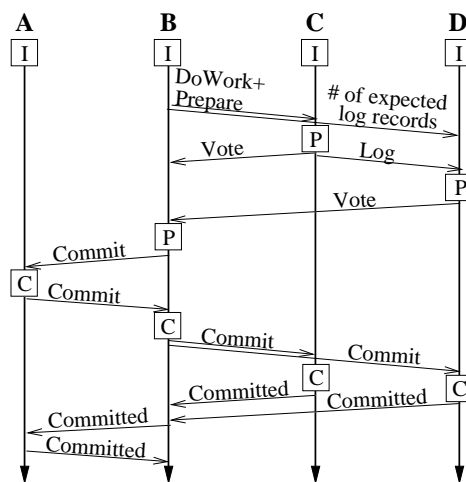


Figure 3.13: The ClustRa Commit Protocol. *B* is the primary coordinator, *A* is the backup coordinator, *C* is the primary cohort and *D* is the backup cohort.

### ClustRa Commit Protocol

ClustRa is a main-memory database system [50]. It uses a novel one-phased commit protocol which is geared towards low delay, and it uses piggybacking of acknowledgement messages. The protocol, CCP, is shown in Figure 3.13. The primary coordinator initiates the protocol by piggybacking *Prepare* on the work request to all primary participants. In addition, it sends a message to each backup participant with the number of expected log records to receive from the corresponding primary. The backup participant then waits until it receives the log records from the primary and then send its vote to the primary coordinator. When the primary coordinator has received votes from all cohorts, both primary and backup, it makes a decision and notifies the backup coordinator. The backup coordinator acknowledges the decision and reply to the primary. The primary sends the decision to all primary and backup participants in parallel. The participants acknowledge the decision. Then, finally the backup coordinator is notified about the successful completion of the transaction.

CCP uses the Early Answer optimization, as explained in 3.1.4, in combination with a reply directly from the backup participant to the primary coordinator. Thus, the client sees a short response time, and the throughput is increased because of piggybacking. However, there are still some unnecessary messages sent, which hampers the performance.

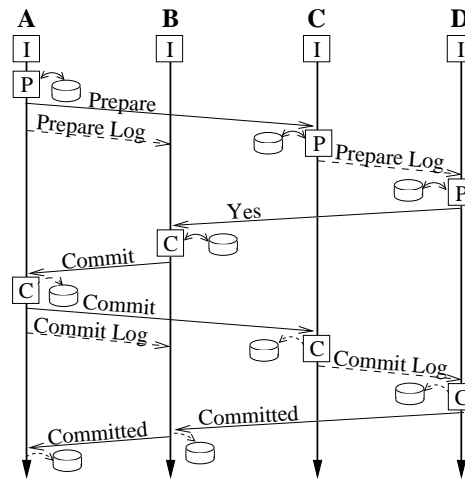


Figure 3.14: The 2-safe Primary Backup Commit protocol. *A* is the primary coordinator, *B* is the backup coordinator, *C* is the primary cohort and *D* is the backup cohort.

### 2PC with Backups

Mehrotra, Hu and Kaplan [68] describe a specially designed commit protocol for primary-backup systems. A successful commit is shown in Figure 3.14: The voting phase is initiated as the primary coordinator, *A*, flushes the *prepare* record to disk. *Prepare* is sent to the primary cohorts (only one is shown in the figure, *C*) and *Prepared* asynchronously to the backup coordinator, *B*. Each primary cohort asynchronously sends the prepare log record to the corresponding backup cohort, *D*. *D* flushes *Prepare* to stable storage and sends *Yes* *B*.

After *B* has collected votes from all participants and received the *Prepare* log record from *A*, it decides *COMMIT*, and flushes the decision to disk. The decision is then sent to *A*, which logs and forwards the decision to the primary cohorts, *C*. In addition, the *Commit* record is asynchronously sent to *B*. *C* logs the decision, releases its locks and asynchronously sends *Commit* to its backup, *D*. *D* appends the *Commit* record to its own log and acknowledges the decision to *B*. After receiving the *Commit* record from *A* and all cohorts have acknowledged the commit, *B* writes an *EndOfTransaction* record to the log, sends *Committed* to *A* and forgets about the transaction. *A* can in turn forget about the transaction after receiving *Committed* from *B* and logging *EndOfTransaction*.

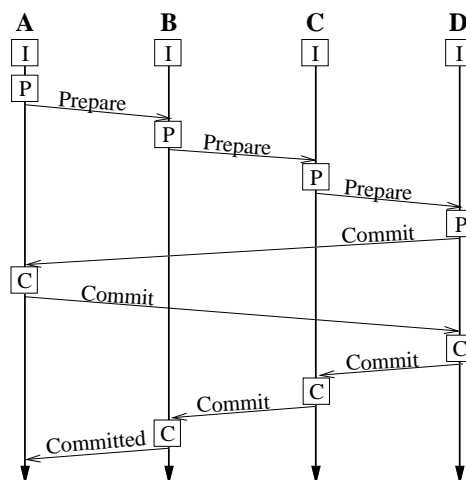


Figure 3.15: The MySQL Cluster Commit protocol. *A* is the coordinator, *B* is the primary cohort, *C* and *D* are backup cohorts.

### RODAIN

The RODAIN database architecture [79, 65] is designed for telecommunication systems. It is a real-time object-oriented architecture of a database management system. It keeps a disk-based backup of the database at the backup node in case both primary and backup should fail at the same time. It presents an approach for committing a single primary-backup pair. The primary sends all log records together with a commit log record to the backup. The backup stores the log in its main memory and acknowledges the receipt. Then the primary can return committed. The backup continues by updating its copy of the database and persistently saving the log to disk. The logs are then removed from main memory, and a message is sent to the primary to do the same.

This approach does not discuss how to do global coordination of commit. In addition, if the acknowledgement and remove-logs messages from the backup are lost, the primary re-sends the commit message. However, the backup may already have removed the log records from main memory and active transaction table. In this case, the backup would have to search for it on disk. As a worst-case scenario, the entire disk would have to be checked.

### 3.2.4 Other Non-Blocking Approaches

#### MySQL Cluster

MySQL Cluster uses a two-phase commit protocol [93], from here on referred to as M CCP. The coordinator is not replicated. To avoid blocking, informa-

tion about all participants are piggybacked on the prepare messages. Should the coordinator fail, this mechanism enables the participants to query the other participants for the status of the in-doubt transactions.

In a failure-free scenario, MCCP is executed in a circular fashion as shown in Figure 3.15. The coordinator *A* initiates the protocol by sending **Prepare** to all primary participants, *B*. *B* forwards its vote to the first backup participant *C*, which forwards it to *D*. *D* sends a prepare message to *A* saying that the participant has prepared. The commit phase is executed in the opposite direction. Thus, *A* can safely release the locks when it receives **Commit** because all backups know the outcome.

### 3.3 Managing Non-Determinism in Replicated Transactional Systems

Replication and transactions have historically been two separate techniques for achieving fault tolerance. FT-CORBA [80] is an example of system treating them separately. Studying two systems, one with transactions and no group communication, and one with group communication and no transactions, Little and Shrivastava [66] concluded that a distributed transaction system can profit from using group communication, especially for supporting fast failover and active replication. Felber and Narasimhan [29] noted that, in particular, stronger consistency and higher availability can be achieved if transactions and replication are integrated.

#### Process Pairs

The concept of *process pairs* [39] was developed in the context of databases and combined with transactions to achieve highly available and highly reliable processes. A process pair is the equivalent of a passively replicated object group. A primary executes requests from the clients and sends state updates to the backups. Heartbeat messages are used to detect a failed primary. When that happens, a protocol to agree on a new primary is executed. This approach assumes deterministic execution and, hence, does not handle the problems related to non-determinism.

Other systems that support transactions in an active replicated environment also exist. GroupTransactions [82] allows transactional servers to be process groups and, thus, provides an integration of the concepts. Circus [20] extends the replicated environment with support for multi-threaded servers through a novel commit and synchronization protocol. Zhao et al. [111]

replicate the transaction coordinators, which leads to a non-blocking 2PC protocol, a highly desirable property since it ensures liveness.

#### Group Communication Primitives

Schiper and Raynal [96] presents an approach where transactions are implemented with group communication primitives. All operations of a transaction are put into one message, which is sent to all process groups participating in the transaction. Because of the total order and atomic delivery [105, pp. 389–391] provided by the group communication system, the transaction will be atomic and serializable, and the replicated nature of the process group will ensure durability. However, this approach assumes that process groups will always be able to complete the operations contained in the message. This is an unreasonable constraint since a process group might be unable to execute all operations because of internal constraints. For example, a banking account may not be overdrawn. In a normal transaction processing system, this causes the process group to vote abort, but since no abort mechanism exists, it may result in an inconsistent system state.

#### Stateless Middle Tiers

Frølund and Guerraoui [32] present a complete integration of replication and transactions for three-tier applications. However, it supports only stateless middle-tier servers, forcing all state to be stored in the end-tier databases.

#### Updating Backups

Systems that support non-deterministic execution must be able to control its effects. ITRA [24] is an approach that handles them by replicating the result of each non-deterministic operation to the backups. Though ITRA claims to support transactions, the exact integration is not presented.

Pleisch et al. [84, 85] describes two schemes to handle non-determinism, one optimistic and one pessimistic. The first allows a subtransaction to be committed before its parent, while the latter forces the subtransaction to wait for the commit of the parent. By sending undo information to the backups before invoking a server, orphan subtransactions can be terminated in the pessimistic case, and compensated in the optimistic case. A CORBA related approach [29] avoids the need for undo information by using a centralized transaction manager and nested transactions. If a parent transaction cannot be completed, the transaction manager aborts all subtransactions. However, this approach does not clarify the intricate details of this abortion.



# Chapter 4

## Conclusions

This chapter concludes the thesis, lists some of the main contributions and limitations to the research and avenues for future work.

### 4.1 Contributions of this thesis

Commit processing has been studied for decades by many researches. However, the performance potential of executing distributed commit processing in main memory only, while persistence is provided by replication, has not been sufficiently addressed. This thesis has emphasized the problems and challenges associated with commit processing in high performance and fault-tolerant main-memory databases. The two research questions posted in Section 1.2 are addressed by the papers in Part II. The questions are repeated here:

**Q-1** *How can commit processing be tailored to main-memory primary-backup transactional systems to improve performance and fault tolerance?*

**Q-2** *How can non-determinism be managed in main-memory primary backup-transactional systems, while adding minimal overhead?*

Papers 2 – 6 go a long way in answering Question Q-1. Tailored protocols with new optimizations have been developed for both single-fault-tolerant systems and  $k$ -fault-tolerant systems, and the results from the evaluations indicate that these protocols are very effective. Compared to the state-of-the-art protocol CCP, the throughput increases by over 50% for transactions consisting of three short update subtransactions. For single tuple transactions, a further improvement of approximately 80 – 130% is possible. Response times are also improved, the relative improvement is, as expected, larger for higher throughputs.

In Paper 2, circular commit protocols are introduced. The correctness and single-fault-tolerance is proven, and a simple analysis shows the potential performance improvement by comparing this method to a standard rewriting of the two-phase commit protocol (R2PC). Evaluations in Paper 3 show that throughput can be improved by approximately 25% by using C1PC instead of CCP. Paper 3 improves on these circular protocols even further by introducing dynamic coordinators. The optimization achieves two things: Reducing the overhead caused by a transaction arriving at a node that is not a participant in the transaction, and load balancing. The dynamic coordination approach is shown to improve the throughput by another 25%. Paper 4 investigates the effects of prioritizing tasks and messages, and reduction in overhead by piggybacking messages. For a 5 millisecond response time limit, it supports 20% more transactions, and, at high throughputs, the overhead is reduced even more.

Some applications require a system to be more than single-fault tolerant. Thus, Paper 5 generalized the circular protocols to multiple backups. When more nodes are added, the overhead increases and the performance decreases. This can be partially alleviated by adding more nodes to the system to distribute the transactional load. For 3 nodes, C1PC tolerate 40% more throughput than R1PC. Paper 6 uses the fact that consistency can be relaxed for certain transactions. In addition, it presents an approach for transactions that require no global commit; i.e., the transaction only accesses data on a single node. In such an environment, this approach improves the possible throughput by 150% for read-heavy and 70% for write-heavy transactions, compared to just using C1PC. Papers 3 – 6 used simulation to evaluate the results. The simulations were validated by performing statistical analysis.

Paper 1 gives an approach that offers a solution to Question Q-2. Orphan requests are removed as a part of commit processing, and only if the primary has failed. There are no extra messages required during normal operation, and the only extra overhead is the piggybacking of a couple of integers on the messages to the transaction manager. Real experiments were performed on a prototype implementation.

Another contribution of this work is an extensive presentation of state-of-the-art in Chapter 3 and in each paper. Chapter 3 gives a thorough walk-through of most commit protocols both for disk-based and main-memory systems.

The practical consequences of this work can be summed up as follows. Main-memory databases should use tailored commit protocols to reduce the overhead added by these protocols. In addition, it is possible to handle orphan requests in a transactional system while adding very little overhead during normal operation. The overhead is limited to a few bytes per message.

## 4.2 Limitations

There are some limitations to this work. First, the work is focused on pure main-memory primary-backup systems. No persistent medium except replication has been used. However, some of the optimizations have merit in general transactional and database systems as well; i.e., dynamic coordinators and prioritizing and piggybacking.

Second, no simulation model incorporates all the aspects of the real world, thus any model is just an approximation. The prototype implementation of the orphan request handling algorithm needs to be able to handle all kinds of failures. Thus, real and complete implementations are still needed for the circular commit protocols, their optimizations and for the algorithm to avoid orphan requests.

Special hardware has not been considered here. The write delay of simple *solid-state drives* (SSDs) is too long, they offer too low write throughput and they must be replicated to support fault tolerance and availability. More advanced SSDs are very expensive at the moment. However, the development in this area is currently very rapid and SSDs can prove to be cost efficient in the future. Battery-driven main memory solves the fault-tolerance problem, but needs replication to achieve high availability.

SQL interactions typically have multiple reply-request interactions with the client before a transaction is committed. In this setting, the proposed solutions do not have much effect. Therefore, this thesis assumes the existence of precompiled transactions or Persistent Store Modules, which limit the interactions with the client to a single request-reply interaction, and hence, the commit processing is a significant part of the total transaction time.

By using main memory instead of disk, shorter delays are achievable. Total throughput, however, is not necessarily increased since I/O operations can be batched and performed in parallel with CPU execution.

## 4.3 Future Work

The following topics are identified as possible directions for further research.

### Implementation into Real Systems

This work has mainly focused on improving the performance of main-memory commit processing and evaluated the approaches using simulations and analyses. While applicable, these results are not as realistic as results from real implementations. Thus, the protocols should be implemented in an existing

main-memory database system, where real measurements of improvement can be obtained. Simulations provides, however, the means to try other configurations than available in today's hardware.

### **Vary the Workloads**

To reduce the number of variables in the simulations, the number and type of subtransactions were held constant. However, to examine the effect of using these algorithms in a real world, different types of loads should be applied to get more results.

### **Extend Prototype of the Orphan Handling System**

The prototype implemented as a part of the Jgroup/ARM system did not handle all types of failures. To be of any practical use, it should be extended to handle more types of failures and restart of failed nodes. Also, to be of any use in real-time system it should be integrated in a high-performance system to achieve shorter response times.

### **Real Experiments During Failures**

In most of this work, the performance have not been evaluated during failures. In a non-simulated environment failures *will* happen, and the system must be able to perform close to optimal even during failure scenarios. Thus, measurements of the performance during failures should also be made.

### **Consistency Relaxations**

The approaches presented in this thesis all offer the consistency criteria of serializability. However, for some applications this requirement is stronger than it has to be. Thus, methods for various consistency relaxations and one-safe approaches should be investigated. By relaxing the consistency requirements, there is a potential performance increase.

## Bibliography

- [1] Maha Abdallah and Philippe Pucheral. A single-phase non-blocking atomic commitment protocol. In *DEXA '98: Proceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 584–595, London, UK, 1998. Springer-Verlag. ISBN 3-540-64950-6.
- [2] Maha Abdallah, Rachid Guerraoui, and Philippe 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. ISBN 0-8186-8603-0.
- [3] Y. J. Al-Houmaily and P. K. Chrysanthis. Two-phase commit in gigabit-networked distributed databases. In *Proceedings of the 8th ISCA International Conference on Parallel and Distributed Computing Systems*, pages 554–560, 1995.
- [4] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.
- [5] Entertainment Software Association. Annual report - 2006, 2007. URL <http://www.theesa.com/>.
- [6] Özalp Babaoglu and Sam Toueg. Understanding non-blocking atomic commitment. Technical Report UBLCS-93-2, Laboratory for Computer Science, University of Bologna, Jan 1993.
- [7] Roberto Baldoni, Carlo Marchetti, and Alessandro Termini. Active software replication through a three-tier approach. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 109–118, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1659-9.
- [8] Bela Ban. JavaGroups - group communication patterns in Java. Technical report, Department of Computer Science, Cornell University, April 1998.
- [9] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, and S. Haldar. DataBlitz storage manager: Main-memory database performance for critical applications. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 519–520, New York, NY, USA, 1999. ACM. ISBN 1-58113-084-8.

- [10] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981. ISSN 0360-0300.
- [11] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publ. Co., Inc., 1987. ISBN 0-201-10715-5.
- [12] Kenneth P. Birman and Robbert Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1993. ISBN 0-818-65341-8.
- [13] Andrea J. Borr. Transaction monitoring in ENCOMPASS: Reliable distributed transaction processing. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 155–165. IEEE Computer Society, 1981.
- [14] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems. In S. Mullender, editor, *Distributed Systems*, ACM Press, chapter 8: The primary-backup approach, pages 199–216. Addison-Wesley, second edition, 1993. ISBN 0-201-62427-3.
- [15] Cellular-News. <http://www.cellular-news.com/story/26851.php>, 2007.
- [16] Tushar Deepak Chandra and Sam Toueg. Time and message efficient reliable broadcasts. In *Proceedings of the 4th international workshop on Distributed algorithms*, pages 289–303, New York, NY, USA, 1991. Springer-Verlag. ISBN 0-387-54099-7.
- [17] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. ISSN 0004-5411.
- [18] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, 1985. ISSN 0734-2071.
- [19] Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Computer*, 22(2):63–70, 1989. ISSN 0018-9162.
- [20] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the tenth ACM symposium on Operating systems principles*, pages 63–78. ACM Press, 1985. ISBN 0-89791-174-1.

- [21] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [22] X. Défago and A. Schiper. Specification of replication techniques, semi-passive replication and lazy consensus. Technical Report IC/2002/007, École Polytechnique Fédérale de Lausanne, Switzerland, February 2002.
- [23] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998.
- [24] Eliezer Dekel and Gera Goft. ITRA: Inter-tier relationship architecture for end-to-end QoS. *The Journal of Supercomputing*, 28:43–70(28), April 2004.
- [25] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-128-8.
- [26] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996. ISSN 0001-0782.
- [27] EVE Online. <http://www.eve-online.com/news/newsOfEve.asp?newsID=505>, 2007.
- [28] fastDB. <http://www.fastdb.org/>, 2007.
- [29] Pascal Felber and Priya Narasimhan. Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE*, pages 737–754. Springer-Verlag, 2002. ISBN 3-540-00106-9.
- [30] Pascal Felber, Rachid Guerraoui, and Andre Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.
- [31] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. ISSN 0004-5411.

- [32] Svend Frølund and Rachid Guerraoui. Transactional exactly-once. Technical report, Hewlett-Packard Laboratories, July 1999.
- [33] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 04(6):509–516, 1992. ISSN 1041-4347.
- [34] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book*. Prentice-Hall, Inc., 2002. ISBN 0-13-031995-3.
- [35] Dieter Gawlick and David Kinkade. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985.
- [36] GigaSpaces. <http://www.gigaspace.com>, 2007.
- [37] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978. ISBN 3-540-08755-9.
- [38] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154. IEEE Computer Society, 1981.
- [39] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1558601902.
- [40] R. Guerraoui and A. Schiper. The decentralized non-blocking atomic commitment protocol. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, page 2, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7195-5.
- [41] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 692–697, Hong Kong, 1996.
- [42] Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972, pages 87–100, Le Mont-Saint-Michel, France, 1995. Springer-Verlag. ISBN 3-540-60274-7.

- [43] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 2nd edition, 1993.
- [44] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983. ISSN 0360-0300.
- [45] Jayant R. Haritsa, Krithi Ramamritham, and Ramesh Gupta. The PROMPT real-time commit protocol. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):160–181, 2000. ISSN 1045-9219.
- [46] S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. MonetDB/X100 at the 2006 TREC TeraByte Track. In *Proceedings of the Text REtrieval Conference (TREC-2006)*, Gaithersburg, MD, USA, November 2006.
- [47] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. ISSN 0164-0925.
- [48] Michael Heytens, Sheralyn Listgarten, Marie-Anne Neimat, and Kevin Wilkinson. Smallbase: A main memory DMBS for high-performance applications (release 3.1). Technical report, Hewlett Packard Laboratories, Palo Alto, CA, USA, 1994.
- [49] Ching-Liang Huang and Victor O. K. Li. A quorum-based commit and termination protocol for distributed database systems. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 136–143, Washington, DC, USA, 1988. IEEE Computer Society. ISBN 0-8186-0827-7.
- [50] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, 1995.
- [51] International Standardization Organization ISO. Information Processing Systems - Database Language SQL. ISO/IEC 9075, 1992.
- [52] ISO. Information Technology - Database Language SQL - part 4: Persistent Stored Modules (SQL/PSM). ISO/IEC 9075-4, 2003.

- [53] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 48–59, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8.
- [54] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Sergio Arévalo. A low-latency non-blocking commit service. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 93–107. Springer-Verlag, 2001. ISBN 3-540-42605-1.
- [55] Heine Kolltveit and Svein-Olaf Hvasshovd. The Circular Two-Phase Commit Protocol. In *Proceedings of International Conference of Database Systems for Advanced Applications*, pages 249–261. Springer-Verlag, 2007.
- [56] Arnas Kupšys. *JMSGroups: JMS Compliant Group Communication*. PhD thesis, Kaunas University of Technology, Lituanie, 2005.
- [57] Arnas Kupšys and Richard Ekwall. Architectural Issues of JMS Compliant Group Communication. In *4th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2005)*, 2005.
- [58] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
- [59] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th Conference on Very Large Databases*. Morgan Kaufman, 1993.
- [60] Butler W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265, London, UK, 1981. Springer-Verlag. ISBN 3-540-10571-9.
- [61] Jean C. Laprie. *Dependability: Basic concepts and terminology in English, French, German, Italian, and Japanese (Dependable computing and fault-tolerant systems)*. Springer-Verlag, December 1992. ISBN 3-211-82296-8.
- [62] Jean-Claude Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Highlights from Twenty-Five Years*, pages 2–11. IEEE Computer Society Press, 1995. ISBN 0-8186-7150-5.

- [63] Inseon Lee and Heon Young Yeom. A single phase distributed commit protocol for main memory database systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 44–51, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1573-8.
- [64] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. An optimistic commit protocol for distributed transaction management. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, pages 88–97, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-425-2.
- [65] Jan Lindström, Tiina Niklander, Pasi Porkka, and Kimmo E. E. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Proceedings of the International Workshop on Databases in Telecommunications*, pages 158–173, London, UK, 2000. Springer-Verlag. ISBN 3-540-67667-8.
- [66] Mark C. Little and Santosh K. Shrivastava. Integrating group communication with transactions for implementing persistent replicated objects. *Lecture Notes in Computer Science*, 1752:238–253, 2000. ISSN 0302-9743.
- [67] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA, October 1995. Workshop held during the 7th IEEE Symp. on Parallel and Distributed Processing, (SPDP-7).
- [68] Sharad Mehrotra, Kexiang Hu, and Simon Kaplan. Dealing with partial failures in multiple processor primary-backup systems. In *CIKM '97: Proceedings of the Sixth International Conference on Information and Knowledge Management*, pages 371–378, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-970-X.
- [69] Hein Meling, Alberto Montresor, Özalp Babaoğlu, and Bjarne E. Helvik. Jgroup/ARM: A distributed object group platform with autonomous replication management for dependable computing. Technical Report UBLCS-2002-12, University of Bologna, October 2002.
- [70] MMOGChart. An analysis of MMOG subscription growth - version 21.0. <http://mmogchart.com/>, 2006.

- [71] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R\* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986. ISSN 0362-5915.
- [72] Seshadri Mohan and Ravi Jain. Two User Location Strategies for Personal Communications Services. *Personal Communications*, 1(1):42–50, 1994.
- [73] Curt Monash. The database technology of Guild Wars. <http://www.dbms2.com/2007/06/09/the-database-technology-of-guild-wars/>, 2007.
- [74] Morningstar. [http://news.morningstar.com/news/ViewNews.asp?article=/DJ/200711192059DOWJONESDJONLINE000644\\_univ.xml](http://news.morningstar.com/news/ViewNews.asp?article=/DJ/200711192059DOWJONESDJONLINE000644_univ.xml), 2007.
- [75] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996. ISSN 0001-0782.
- [76] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A fault tolerance framework for {CORBA}. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 150, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0213-X.
- [77] MySQL Cluster. <http://www.mysql.com/products/database/cluster/>, 2007.
- [78] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990. ISSN 0196-6774.
- [79] T. Niklander, J. Kiviniemi, and K Raatikainen. A real-time database for future telecommunication services. In D Gaiti, editor, *Intelligent Networks and Intelligence in Networks*. Chapman & Hall, 1997.
- [80] OMG. *Fault Tolerant CORBA*. Object Managment Group, March 2004. OMG Technical Committee Document formal/04-03-21.
- [81] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979. ISSN 0004-5411.

- [82] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Group transactions: An integrated approach to transactions and group communication. In *Workshop on Concurrency in Dependable Computing*, Newcastle Upon Tyne, United Kingdom, 2001.
- [83] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980. ISSN 0004-5411.
- [84] S. Pleisch, A. Kupšys, and A. Schiper. Preventing orphan requests in the context of replicated invocation. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 119 – 128, Florence, Italy, October 2003. IEEE.
- [85] S. Pleisch, A. Kupšys, and A. Schiper. Replicated invocations. Technical report, Swiss Federal Institute of Technology (EPFL), September 2003.
- [86] Stefan Poledna. Replica determinism in distributed real-time systems: A brief survey. Research Report 6/1993, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1993.
- [87] Polyhedra. <http://www.enea.com/polyhedra>, 2007.
- [88] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Number 818/2252 in Research Reports, ESPRIT 818/2252. Springer-Verlag, 1991. ISBN 3-540-54985-4.
- [89] Michael Rabinovich and Edward D. Lazowska. A fault-tolerant commit protocol for replicated databases. In *PODS '92: Proceedings of the eleventh ACM SIGMOD symposium on Principles of Database Systems*, pages 139–148, New York, NY, USA, 1992. ACM Press. ISBN 0-89791-519-4.
- [90] P. Krishna Reddy and Masaru Kitsuregawa. Blocking reduction in two-phase commit protocol with multiple backup sites. In *DNIS '00: Proceedings of the International Workshop on Databases in Networked Information Systems*, pages 200–215, London, UK, 2000. Springer-Verlag. ISBN 3-540-41395-2.
- [91] P. Krishna Reddy and Masaru Kitsuregawa. Reducing the blocking in two-phase commit protocol employing backup sites. In *Conference on Cooperative Information Systems*, pages 406–416, 1998.

- [92] Reuters. <http://investing.reuters.co.uk/news/articleinvesting.aspx?type=media&storyID=nL29172095>, 2007.
- [93] Mikael Ronström. High Availability features of MySQL Cluster. White Paper, MySQL AB, 2005.
- [94] Mikael Ronström. *Design and Modelling of a Parallel Data Server for Telecom Applications*. PhD thesis, Uppsala University, 1998.
- [95] Kurt Rothermel and Stefan Pappé. Open commit protocols for the tree of processes model. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 236–244. IEEE Computer Society Press, 1990.
- [96] André Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, 1996. ISSN 0001-0782.
- [97] Fred B. Schneider. Replication management using the state machine approach. In *Distributed systems (2nd Ed.)*, pages 169–197. ACM Press/Addison-Wesley Publishing Co., 1993. ISBN 0-201-62427-3.
- [98] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990. ISSN 0360-0300.
- [99] Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142. ACM Press, 1981. ISBN 0-89791-040-0.
- [100] solidDB. <http://soliddb.com/>, 2007.
- [101] Peter M. Spiro, Ashok M. Joshi, and T. K. Rengarajan. Designing an optimized transaction commit protocol. *Digital Technical Journal*, 3(1):70–78, 1991. ISSN 0898-901X.
- [102] James W. Stamos and Flaviu Cristian. A low-cost atomic commit protocol. In *Proceedings of the Ninth Symposium of Reliable Distributed Systems*, 1990.
- [103] James W. Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4), 1993. ISSN 0926-8782.

- 
- [104] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.
- [105] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001. ISBN 0130888931.
- [106] TimesTen. <http://www.timesten.com/>, 2007.
- [107] Alexey Vaysburd and Ken Birman. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems*, 4(2):71–80, 1998. ISSN 1074-3227.
- [108] P Veríssimo, P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton. The Extra Performance Architecture (XPA). In D. Powell, editor, *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, ESPRIT Research Reports, chapter 9, pages 211–266. Springer Verlag, nov 1991.
- [109] M. Wiesmann. *Group Communications and Database Replication: Techniques, Issues and Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, May 2002. Number 2577.
- [110] WoW. World of warcraft reaches new milestone: 10 million subscribers. <http://blizzard.co.uk/press/080122.shtml>, 2008.
- [111] W. Zhao, L. E. Moser, and P.M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 290–297, 2002.



**Part II**  
**Publications**



# Paper I

Preventing Orphan Requests by Integrating Replication and Transactions.

Heine Kolltveit and Svein-Olaf Hvasshovd.

In *Advances in Databases and Information Systems*,  
SpringerLink.

September 29 - October 3, 2007, Varna, Bulgaria.



# Preventing Orphan Requests by Integrating Replication and Transactions

Heine Kolltveit and Svein-Olaf Hvasshovd

Department of Computer and Information Science  
Norwegian University of Science and Technology  
NO-7491 Trondheim, Norway

## Abstract

Replication is crucial to achieve high availability distributed systems. However, non-determinism introduces consistency problems between replicas. Transactions are very well suited to maintain consistency, and by integrating them with replication, support for non-deterministic execution in replicated environments can be achieved. This paper presents an approach where a passively replicated transaction manager is allowed to break replication transparency to abort orphan requests, thus handling non-determinism. A prototype, implemented using existing open-source software, Jgroup/ARM and Jini, has been developed, and performance and failover tests have been executed. The results show that while this approach is possible, components specifically tuned for performance must be used to meet real-time requirements.

## I.1 Introduction

Fault tolerance is an important property of real-time and high-availability applications. By moving from a centralized to a distributed system, the probability of a total system failure decreases, while the probability of a partial failure increases. A partial failure that is not dealt with correctly could easily jeopardize both consistency and availability of a system.

The *availability* of a system is defined as the fraction of the time that the system performs requests correctly and within specified time constraints [10], while *consistency* is the property that guarantees that the system will behave according to the functional requirements, and applies both to internal

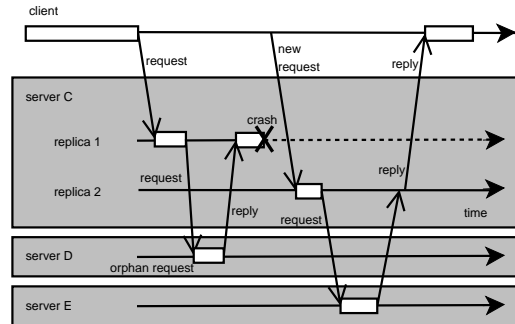


Figure I.1: An orphan request caused by non-determinism in server A

(state changes) and external (output) behavior. Traditionally, transactions are used to ensure consistency [10], while replication provides availability [11]. The two most common types of replication are passive [3] and active [24].

A replicated system must be able to handle the problems occurring due to replicated invocations. A *replicated invocation* is a request from a replicated client to a (possibly replicated) server [20]. The actual problem and solution depends on whether the client is deterministic or not. For deterministic clients, it is only a matter of detecting duplicates, making sure that each request is only executed once and returning the same answer to all duplicate requests. Many clients, however, are not deterministic. Two common sources for non-determinism are multi-threading and timeouts, but others also exist [22]. These clients are said to be *non-deterministic* and the problem of orphan requests may arise if one of them crashes.

An *orphan request* is a request that is received and processed by a server, but it is no longer valid, normally because of a client failure. Figure I.1 illustrates an orphan request. Replica  $A_1$  receives a request from the client and invokes service  $B$ .  $A$  is then said to be a *client* of server  $B$ .  $A_1$  then fails before it can reply to the client. Replica  $A_2$  is chosen as the new primary and receives the retransmitted request from the client. However, since service  $A$  is non-deterministic, the re-sent request 1 might not lead to an identical request 2 to be sent to  $B_1$ . Consequently, the request from  $A_1$  to  $B_1$  is an orphan request and its results must be removed. If such a request is not handled, it may cause inconsistencies. Even worse, they might spread to other parts of the system, making the whole system inconsistent.

The main contribution of this paper is an integration of transactions and replication where possible orphan requests caused by non-determinism are aborted. Standard atomic commitment protocols, like 2-Phase Commit (2PC) [9], can not guarantee to remove all orphan requests. The approach

suggested by the authors solves this by allowing the transaction manager to break replication transparency and therefore see the individual replicas of the transaction participants instead of the whole replica group. As long as at least one replica of the transaction manager is available it also renders 2PC non-blocking [10, 23]. The problem of orphan requests in replicated systems have been handled before by integrating transactions and replication (e.g. [20, 6, 8]), but these approaches are either ineffective (extra messages in the critical path of the transaction), do not support state in all tiers or make unrealistic assumptions regarding the detection of orphans. The approach adopted in this paper does not have these weaknesses and, at the same time it, supports checkpointing at any time and restart of crashed replicas.

A prototype based on existing open-source group communication, Jgroup/ARM [16], and transaction implementations, Jini Transaction Service [25], is also presented. The prototype is performance evaluated to see if it can meet the stringent requirements of real-time systems.

The rest of this paper is organized as follows: The system model is presented in Section I.2. Section I.3 describes other approaches related to the integration of replication and transactions, while supporting non-determinism. A detailed description of how the integration is performed is given in Section I.4. The method developed in Section I.4 has been implemented in a test system and the tests performed on this system are presented in Section I.5 and discussed in Section I.6. Finally, Section I.7 concludes the paper.

## I.2 System Model

The system consists of a set  $S$  of fail-crash processes connected through unreliable channels without network partitions. The processes or *nodes* communicate by sending messages. A group  $G$ , which is a subset of  $S$ , implements a service that can be invoked by clients. These may be replicated. A node in  $G$  is called a member or *replica* of that service. Group membership is controlled by a *group membership service* that provides an interface for changes in  $G$ , implements a failure detector, notifies members of changes in  $G$  and controls that a request is sent to the correct replica(s) [4]. At any given time, members of a group have a *view* of the group which is the set of the agreed-upon members. Any replica that crashes is eventually excluded from the view, and any restarted and recovered replica is eventually included in the view.

Passive replication is used, i.e. for each group, there is a primary replica that receives, processes and replies to requests. The state of the backups are updated by periodically performing checkpoints [10], while forced writes

of log records are propagated as a part of the atomic commitment protocol. The most common atomic commitment protocol, 2PC, is used.

Replicas are stateful, but have no persistent state. If a replica crashes and is restarted, the state is retrieved from one of the other replicas of the same group. The approach here assumes that there is always at least one replica that has not crashed, therefore the state will never be completely lost. Because there is no persistent state, it is not possible to distinguish a restarted node from a new node. All replicas are assumed to be non-deterministic, thus they can all produce orphan requests.

A request is assumed to be eventually received. They are periodically retransmitted from clients until a reply is received, and duplicates are filtered at each server. Thus, if a primary replica fails and a view with a new primary is installed, the new primary will receive the request when it is re-sent.

### I.3 Related Work

Replication and transactions have historically been two separate techniques for achieving fault tolerance. For instance, CORBA's transaction service (OTS) [18] and replication service (FT-CORBA) [19] are not integrated. A study by Little and Shrivastava [13] looks at two systems, one with transactions and no group communication, and one with group communication and no transactions. Their conclusion is that group communication can be useful for transactions, especially for supporting fast fail-over and active replication.

Many projects deal with replication and transactions, but only a few of these present a proper integration of the two concepts in a non-deterministic environment. Systems that support non-deterministic execution must be able to control its effects. ITRA [5] is an approach that handles the effects by replicating the result of each non-deterministic operation to the backups. ITRA supports replicated transactions by replicating the start, join operations, prepare (including all operations), commit and abort operations. However, this is not an optimal integration since it incurs an unnecessary high overhead. In our approach, only prepare, commit and abort operations are replicated.

Frj̄lund and Guerraoui [7] present a complete integration of replication and transactions for three-tier applications. However, their approach supports only stateless middle-tier servers, forcing all state to be stored in the end-tier databases.

Pleisch et al. [20, 21] describes two schemes to handle non-determinism; one optimistic and one pessimistic. The first allows a subtransaction to be committed before its parent, while the latter forces the subtransaction to wait

for the commit of the parent. By sending information about how to undo the changes to the backups before invoking a server, orphan subtransactions can be terminated in the pessimistic case and compensated in the optimistic case. This inserts, however, extra messages in the critical path during failure-free execution.

A CORBA-related approach [6] restarts execution of a failed subtransaction on a backup and aborts subtransactions where a parent transaction has failed. This integration, however, assumes that standard distributed commit protocols can be used and does not handle the intricate details of transaction completion in failure scenarios.

## **I.4 Integration of Transactions and Replication**

This section presents an integration of replication and transactions. The goal is to support non-deterministic execution with minimal overhead caused by the integration in a failure-free scenario and minimal change in the application servers (transaction participants).

### **I.4.1 Replicating the Transaction Manager**

To ensure availability, all single points of failure must be avoided. This is especially important for the transaction manager (TM) because it is a central component involved in distributed transactions. If the TM becomes unavailable, the most widely used atomic commitment protocol, 2PC, may block. By using replication, 2PC becomes non-blocking [10, 23].

The most important job of the TM is to make the decision to unilaterally abort or commit each transaction. Such a highly critical decision does not favor active replication, since every replica will have to behave deterministically. In practice, TMs are non-deterministic since they rely on timeouts in failure scenarios. This adds non-determinism since it cannot be guaranteed that all replicas timeout at exactly the same time [22]. Also, active replication does not scale well since executing the same processes on every replica wastes resources which could have been used to serve other requests.

As can be seen in Figure I.2, the protocol for a passively replicated TM is the same as for the non-replicated before the atomic commitment protocol is initiated by the client in Figure I.3. However, a TM that supports 2PC must be able to persistently store the decision to commit or abort the transaction as the final part of the prepare phase [10, 2, 14]. In a non-replicated environment the decision is made persistent by force-writing a record to disk.

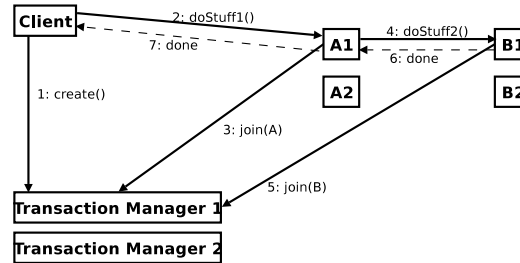


Figure I.2: The execution and join phase of a transaction

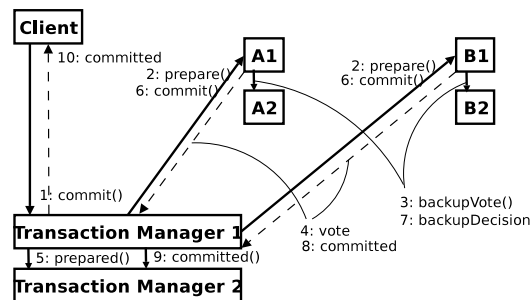


Figure I.3: The successful termination of a transaction

The round-trip transmission time may be a lot shorter than the time needed to write to the disk. A solution where the prepare decision is persistently saved by sending it to the backups (message 5 in Figure I.3) is faster and therefore preferred. In addition, it gives better availability since the prepared transactions can be committed by the backup in case of a primary failure. If a local disk was used, currently prepared transactions may be blocked until the TM has recovered.

A “transaction completed” message is sent to the backups, as indicated by message 9 in Figure I.3. This is done instead of the lazy write to the log in the normal non-replicated 2PC [14]. Hence, the replicated nature of the TM is used to provide both availability and persistence of the decision.

## I.4.2 Replicating the Transaction Participants

The transaction participants should be replicated for the same reason as any other component of a distributed system; to avoid single points of failure. To be able to handle non-determinism, passive replication is used.

Passive replication is subject to fail-overs. A *fail-over* happens when a primary replica fails and another replica of the same service is elected as the new primary by the group membership service. Consequently, any re-sent or new requests will be handled by the new primary. If any operation has been

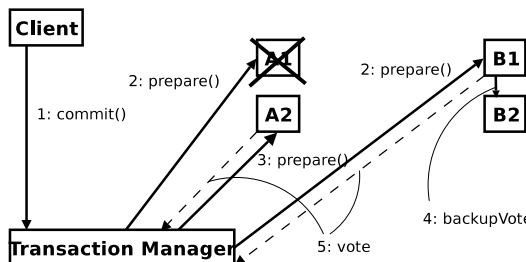


Figure I.4: A failure of a primary, and the consequent fail-over

executed after the previous checkpoint, the new primary might not be fully updated, and care must be taken to avoid inconsistencies caused by orphan requests. Figure I.4 shows a fail-over of a prepare request from  $A_1$  to  $A_2$ .

There are only two ways to cause orphan requests. A failure of the client of the transaction, or a failure of a primary server which acts as a client to another server. The effects of an orphan request can be guaranteed to be removed by aborting all transactions that interact with a replica or client that fails. If the client of the transaction fails, the TM will not receive a commit message from the client. Without the commit message, it can use a timeout to safely abort the transaction. If a primary fails, the TM may still receive a commit message. However, if the TM can determine whether a transaction has been caught in a fail-over or not, it can abort potential orphan requests. The problem is then reduced to detecting failed primary participants of the transaction.

Normally, replication of a server is hidden from the clients of that service, i.e. *replication transparency*. The unpredictable effects of non-determinism, however, can be controlled at the loss of replication transparency for the TM, by sending the prepare message to the primary only. If a participating primary replica of an active transaction fails, the TM can abort the transaction. Note that it is only the prepare message that does not fail-over. Since the primary persistently stores the vote to the backups during the prepare phase, the backups are then updated and an abort or commit message is allowed to fail-over.

Intuitively, by sending the prepare message only to the primary replicas that joined the transaction, primary failures should be detected: Failed primaries will not be able to reply and the transaction is aborted and possible orphan requests are rolled-back by the transactional abort mechanism. This is true for single fail-overs. Figure I.5 illustrates this: When the TM does not get a reply from the failed primary replica,  $A_1$ , it eventually times out and aborts the transaction (as indicated by arrow 5). Contrast this to Figure I.4 where the prepare request is allowed to fail-over and the orphan request to

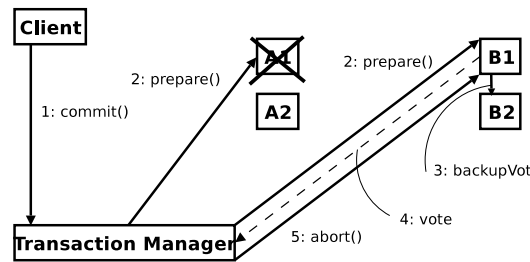


Figure I.5: A failure of a primary, without the fail-over

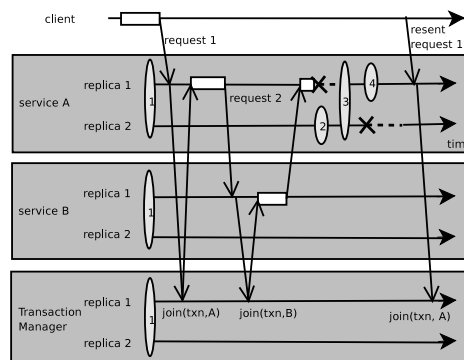


Figure I.6: A double fail-over

service *B* is not handled.

This protocol has one flaw: If the crashed replica is restarted and a second fail-over back to the original primary occurs, the TM may not be able to notice the failures. This is illustrated in Figure I.6. The grey ovals are the current view of the group and the numbers inside are the view identifier. Two fail-overs of group A cause the TM not to notice the fail-over, since the last join message will be identical to the first and looks like a re-send due to a communication error. Therefore, request 2 is an orphan and the transaction should have been aborted. The TM can, however, see the difference of the two join messages if the view identifier was piggybacked on the join message ( $\text{join}(\text{txn}, \text{group}, \text{viewID})$ , instead of  $\text{join}(\text{txn}, \text{group})$ ). In the example, the two join messages would have viewID 1 and 4, respectively. Hence, the new protocol will be able to resolve double fail-overs correctly as well.

If a checkpoint was taken after the first request of the transaction, the new primary is aware of the transaction after a fail-over. If the prepare message was sent to the new primary, orphan requests may not be correctly aborted. Figure I.7 illustrates this. A checkpoint is taken after  $A_1$  has joined the transaction. The checkpoint does not include information about request 2, which becomes an orphan request after replica  $A_1$  fails. If the TM sends

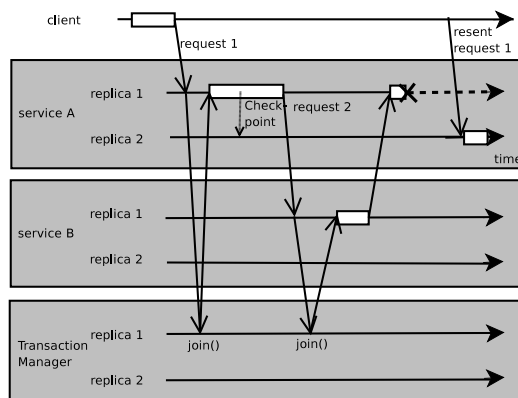


Figure I.7: A checkpoint and a possible orphan request in service  $B$

a prepare message to replica  $A_2$  it would be able to vote yes, and the effects of the orphan request to  $B_1$  could cause inconsistencies.

By simply allowing the TM to break replication transparency and therefore be able to avoid the automatic fail-over of the prepare message to a replicated service, it provides a way for orphan requests to be handled easily and correctly. This approach allows checkpointing at any time, thus reducing the time required to bring the state of the new primary up-to-date and only log records need to be shipped as a part of 2PC. The only requirement for server applications is to implement the 2PC interface. The underlying system adds the view identifier.

### I.4.3 Transaction Termination in Failure Scenarios

When the TM is passively replicated as presented in Section I.4.1, a transaction may be unable to terminate, thereby blocking other transactions from completing. Consider the following case: A transaction has been created and some or all of the participants have joined it. Then the primary TM fails before the prepare phase has completed. This will leave the new primary with no knowledge of the transaction. When the client asks the TM to commit the transaction, the TM will reply that the transaction is unknown, and the client will assume that it has aborted. However, the transaction participants will still hold their locks on the items accessed by the transaction. Without proper termination of these transactions, the locks could be held forever, blocking other transactions from completing.

The locks held by a failed transaction can be removed by the client, if it keeps control of the participants accessed by each transaction. Thus, when the client gets a reply from the TM that the transaction tried to commit is unknown, the client can abort it. Because of possible nested invocations,

each participant must be able to tell which other participants it has caused to join the transaction, and so on. However, if one of the participants also fails, the participants invoked by that participant do not get the abort message.

A better way to remove the locks is to use a timeout. Each participant can periodically poll the TM to get the status of each active transaction. If the TM replies that the transaction is unknown, the transaction can be safely aborted. Also, this takes care of the scenario where the client fails.

This approach causes transaction commitment to be non-blocking as long as at least one of the TMs is available. When combined with the avoidance of fail-over for the prepare message and piggybacking the view identifier, all failures are correctly handled to avoid inconsistencies. Potential orphan requests are rolled back and all types of non-determinism are supported.

## I.5 Implementation and Testing

This section gives an overview of the prototype implementation, as well as the environment used for testing and the results of tests executed on the prototype. A presentation of the prototype is given in Section I.5.1 and the environment for testing and the tests executed are presented in Section I.5.2.

### I.5.1 Prototype Implementation

A prototype of the transaction manager that does not allow fail-over of the prepare request was implemented, along with the transaction participants. The servers were implemented as Jini [1] service, and replicated using the Jgroup/ARM system [16].

The system where the tests are executed consists of four conceptual entities: A client, a transaction manager (TM), and two different banks. The banks and the transaction manager are implemented as Jini services that can be discovered and registered by the Jini registry, *Reggie* [25], or the group-enabled registry, *Greg* [17]. The transaction manager is based on the non-replicated *Mahalo* [25] as well as the actively replicated *Gahalo* [15].

### I.5.2 The Test Environment

Figure I.8 shows the system model. The grey ovals represent entities, while the white boxes are nodes where replicas of servers or the client execute. A single physical node may execute more than one service. The arrows in the figure represent the direction of the invocations.

The life cycle of the transaction used for testing is as follows:

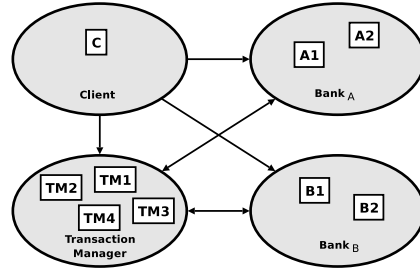


Figure I.8: A model of the system used for testing

1. A transaction is initiated by the client and created by the TM.
2. The client invokes the withdraw operation of Bank<sub>A</sub>, which joins the transaction.
3. The client invokes the deposit operation of Bank<sub>B</sub>, which joins the transaction.
4. The client initiates 2PC, which is controlled by the TM.

As modeled in the figure, the TM can have up to four replicas, and the two banks can have up to two replicas each. These limitations are due to the fact that there were only five nodes available for executing the tests.

A Dual AMD MP 1600+ running at 1.4GHz powered each node. A 100Mbit Ethernet connected them and each had 1024 MB of RAM. The tests were executed using Linux kernel 2.6 and Java version 1.5.0.

All tests were carried out by executing 500 transactions and measuring the elapsed time at the client between transaction initiation and transaction completion. This is referred to as the *response time* of a transaction. Similarly, the response time of an invocation is the time passed between calling the remote method of the client and the return of the method call.

Failure-free performance tests were performed using the following configurations:

1. A non-replicated transaction manager and non-replicated banks.
2. Two passively replicated transaction managers and non-replicated banks.
3. Three passively replicated transaction managers and non-replicated banks.
4. Four passively replicated transaction managers and non-replicated banks.
5. A non-replicated transaction manager, with two replicas of each bank.

Test run	Description	Average (ms)	Standard Deviation (ms)	Delay (%)
<i>Non-replicated system</i>				
1	1 TM and 2 banks	47	10	0
<i>Passive replication of the TM</i>				
2	2 passive TMs and 2 banks	77	17	64
3	3 passive TMs and 2 banks	92	19	96
4	4 passive TMs and 2 banks	106	21	126
<i>Fully replicated system</i>				
5	1 TM and 2x2 banks	75	16	60
6	2 passive TMs and 2x2 banks	148	25	215
7	3 passive TMs and 2x2 banks	164	27	249

Table I.1: A summary of the response times for the test runs in Section I.5.2

6. Two passively replicated transaction managers, with two replicas of each bank.
7. Three passively replicated transaction managers, with two replicas of each bank.

In addition, the response times during fail-overs were measured.

## I.6 Comparing the Test Results

Table I.1 summarizes the results of the test runs made in Section I.5.2. The response time averages and standard deviations are presented<sup>1</sup>. It should be noted that these numbers only apply for these test runs and they should not be interpreted as any general response time guarantee, but rather as properties of the specific test run. However, they can be used as a reference for comparisons between the individual test runs.

The response time of a transaction was measured at the client and is the time elapsed from transaction creation to transaction termination.

The following sections presents a summary of the results from the test runs and compares passive replication and the non-replicated case. Finally, the fail-over delay is examined.

<sup>1</sup>The first 50 transactions of each test run are disregarded in this discussion because of extra startup cost.

### I.6.1 Cost of Replication

Replication increases the overhead of a service. The results of test runs 1–4, as presented in Table I.1, clearly support this assumption. The average response time degrades when adding more replicas of a transaction manager, and the variance of the results increase. The numbers seem to indicate that replicating the TM causes about 50 percent longer response times, while each added replica on top of that increases the response time of about 30 percent of the non-replicated case.

The standard deviation seems to change similarly to the average response time. Table I.1 shows a significant leap for the deviation of the response times when the TM is replicated and then scales linearly when adding the third and fourth replica.

Replication of the transaction participants (test run 5) has similar effects as when only replicating the transaction manager (test run 2). There is a 50 percent increase in the response time and about the same for the standard deviation. For test runs 6 and 7, the overhead increases more. Replicating the TM as well (test run 6) doubles the average response time. The cost of executing a fully replicated system with 2 replicas of each server is three times higher than executing a non-replicated one. If 3 replicas of the TM are executed (test run 7), the response time is three and a half times higher than in the non-replicated case.

A closer inspection reveals that for the fully replicated case (test run 6) the group management threads and layers causes an overhead that delays around 60 percent of the invocations, usually for around 10 – 20 ms. The average delay for a transaction just by running the group management threads was found to be 40 – 50 ms. The time to update the backups was found to be around 22 ms for the commit decision at the TM and 12 ms for each of the other updates. When added together these contribute to an average response time of 150 ms which differs with only 1.3% from the measured total time.

To make the non-replicated case fault-tolerant, the log could be force-written to disk instead of updating the backups. To force-write a record to the disk takes approximately 20 ms. A successfully committed transaction requires three log forces and one lazy log write as part of 2PC [14]. Thus, the completion time for a fully fault-tolerant non-replicated system has a response time of around 107 ms. However, this solution does not provide high availability, since it has single points of failure.

### I.6.2 Fail-over Delay

The observed client-side fail-over delay for the transaction test was found to be as much as 360–490 ms. However, the fail-over delay for a simpler application running on top of the same system was found to be between 200 and 250 ms. These measurements are closer to the real time between a failure and the continuation of the service by a new primary.

Gray and Reuter [10] distinguish five classes of transaction-oriented computing, with various properties and requirements. According to this classification the fail-over delay found here will be sufficient for batch processing, time-sharing (not widely used anymore), client-server and transaction-oriented processing. The last class, real-time processing, however, will probably require client-observed fail-overs of less than 200 ms, depending on the application.

## I.7 Conclusion and Further Work

Many applications require high availability and strong consistency. Since system components fail from time to time, a system must be able to tolerate faults. Well known fault-tolerance techniques include transactions and replication. They are widely used and extensively studied as separate concepts and their efficiency has been well proven. However, to support both availability and consistency in a non-deterministic environment, the techniques should be integrated.

This paper addresses the issue of integrating replication and transactions without enforcing replica determinism. This is a highly desirable property since it allows any kind of application to be built on top of the system. The main contribution is an approach where the transaction manager is allowed to break replication transparency to ensure that no orphan requests survive. Thus, full support for non-determinism in general is achieved.

Tests were performed on a prototype built using existing open-source software. The tests show that transactions can be executed in a passively replicated environment with a 200 percent response time increase. Also, a failure of the primary will cause a fail-over delay of about 400 ms on average for the transaction manager. Measurements on a smaller application, however, indicate that the real fail-over time is probably closer to 200 ms. While these results show that the approach is possible, real-time systems have more stringent performance demands. The response time for a transaction in this prototype is too large for most real-time systems, e.g. telecommunications [12]. However, this is a property of this specific implementation and not

the approach, since both Jini and Jgroup are not tuned for real-time performance. For real-time systems, the entire implementation should be focused on performance and the use of existing open-source software may not be suitable. Also, other transaction models (e.g. hierarchic [14]) than a centralized transaction manager receiving join-messages from all participants should be investigated.

For a real world application, the advantage of increased availability must be weighed against the cost of replication. If the system cannot tolerate the downtime caused by a restart of a machine, replication should be used. On the other hand, if the increased response time cannot be tolerated, but a few minutes of unavailability once in a while can be, replication should not be used.

The system developed in this paper is a prototype where several shortcuts have been made to get a working system for basic testing. To be of any practical use, it must be able to restart crashed replicas, initiate new ones and update the new replicas with the current state. The Jgroup/ARM system has support for automatically performing these actions, but it has not yet been implemented in this prototype. Also, the system must be able to handle all failure scenarios during 2PC to be able to terminate all transaction in the presence of failures.

## References

- [1] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath. *The Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., second edition, 2001.
- [2] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publ. Co., Inc., 1987.
- [3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems. In S. Mullender, editor, *Distributed Systems*, ACM Press, chapter 8: The primary-backup approach, pages 199–216. Addison-Wesley, second edition, 1993.
- [4] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems (3rd ed.): Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., 2001.
- [5] Eliezer Dekel and Gera Goft. ITRA: Inter-tier relationship architecture

- for end-to-end QoS. *The Journal of Supercomputing*, 28:43–70(28), April 2004.
- [6] Pascal Felber and Priya Narasimhan. Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE*, pages 737–754. Springer-Verlag, 2002.
- [7] Svend Frølund and Rachid Guerraoui. Transactional exactly-once. Technical report, Hewlett-Packard Laboratories, July 1999.
- [8] Svend Frølund and Rachid Guerraoui. Implementing e-transactions with asynchronous replication. *Dependable Systems and Networks*, pages 449–458, June 2000.
- [9] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.
- [10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [11] Abdelsalam A. Helal, Bharat K. Bhargava, and Abdelsalam A. Heddaya. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996.
- [12] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, 1995.
- [13] Mark C. Little and Santosh K. Shrivastava. Integrating group communication with transactions for implementing persistent replicated objects. *Lecture Notes in Computer Science*, 1752:238–253, 2000.
- [14] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R\* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.
- [15] Rohnny Moland. Replicated transactions in Jini. Master’s thesis, University of Stavanger, July 2004.
- [16] Alberto Montresor. *System Support for Programming Object-Oriented Dependable Application in Partitionable Systems*. PhD thesis, University of Bologna, Italy, March 2000. Technical Report UBLCS-2000-10.

- 
- [17] Alberto Montresor, Renzo Davioli, and Özalp Babaoğlu. Jgroup: Enhancing Jini with group communication. In *Proceedings of the ICDCS Workshop on Applied Reliable Group Communication*, April 2001.
  - [18] Object Management Group. *Transaction Service Specification*, September 2003. OMG Technical Committee Document formal/03-09-02.
  - [19] Object Management Group. *Fault Tolerant CORBA*, March 2004. OMG Technical Committee Document formal/04-03-21.
  - [20] S. Pleisch, A. Kupšys, and A. Schiper. Preventing orphan requests in the context of replicated invocation. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 119 – 128, Florence, Italy, October 2003. IEEE.
  - [21] S. Pleisch, A. Kupšys, and A. Schiper. Replicated invocations. Technical report, Swiss Federal Institute of Technology (EPFL), September 2003.
  - [22] Stefan Poledna. Replica determinism in distributed real-time systems: A brief survey. Research Report 6/1993, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1993.
  - [23] P. Krishna Reddy and Masaru Kitsuregawa. Reducing the blocking in two-phase commit protocol employing backup sites. In *Conference on Cooperative Information Systems*, pages 406–416, 1998.
  - [24] Fred B. Schneider. Replication management using the state machine approach. In *Distributed systems (2nd Ed.)*, pages 169–197. ACM Press/Addison-Wesley Publishing Co., 1993.
  - [25] Sun Microsystems Inc. *Jini Technology Core Platform Specifications*, 2.1 edition, October 2005.



# Paper II

The Circular Two-Phase Commit Protocol.  
Heine Kolltveit and Svein-Olaf Hvasshovd.  
In *Advances in Databases; Concepts, Systems and  
Applications*, SpringerLink  
April 9 - 12, 2007, Bangkok, Thailand.



# The Circular Two-Phase Commit Protocol

Heine Kolltveit and Svein-Olaf Hvasshovd

Department of Computer and Information Science  
Norwegian University of Science and Technology  
NO-7491 Trondheim, Norway

## Abstract

Distributed transactional systems require an atomic commitment protocol to preserve atomicity of the ACID properties. However, the industry leading standard, 2PC, is slow and adds a significant overhead to transaction processing. In this paper, a new atomic commitment protocol for main-memory primary-backup systems, C2PC, is proposed. It exploits replication to avoid disk logging and performs the commit processing in a circular fashion. The analysis shows that C2PC has the same delay as 1PC, and reduces the total overhead compared to 2PC.

## II.1 Introduction

Main memory prices have dropped significantly over the last years, and the state of many applications and databases can now be fitted entirely in main memory. To make the state both persistent and available, it can be replicated instead of written to disk. For instance, a backup replica (backup for brevity) takes over the processing if a primary replica (primary for brevity) fails. A backup is kept up to date by receiving the same operations as the primary (active replication [25]) or log records from the primary (passive replication [4]). The backup can either apply the log records to its own state or periodically receive checkpoints from the primary. Assuming that the *mean time to fail*, MTTF, is orders of magnitude larger than the *mean time to repair*, MTTR, the system only needs to be single-fault tolerant to completely avoid the need for disk accesses. MTTR can be made very short by employing on-line self-repair mechanisms [13]. In addition, since disk accesses are slow compared to both RAM accesses and network latencies, replication can result in an improvement in performance.

A transaction is a collection of operations that transfers a system reliably from one state to another, while providing the ACID properties [11]: *Atomicity, consistency, isolation* and *durability*. Commonly, transaction termination and atomicity are satisfied by an *atomic commitment protocol*, ACP. The ACP has been shown to be an important factor of total transaction processing time and, in particular, the current industry leading standard, the Two-Phase Commit protocol, 2PC [7], is slow [27, 15, 12]. The delay caused by two rounds of messages and multiple log records flushed to disk cause a significant overhead. Also, a failure of the coordinator might block the participants from completing a transaction [8, 3].

ACP performance and resilience to failures is a well established research field, but optimizations that will have significant effect are still possible under a parallel and replicated paradigm. Thus, this paper presents an ACP called Circular Two-Phase Commit protocol, C2PC. It is an optimized version of 2PC for primary-backup systems. The protocol takes advantage of replication to trade costly flushed disk writes for cheaper message sends and RAM accesses. The idea is to send the vote and decision to the backup instead of a disk. This provides availability for the transaction participants and coordinator and renders 2PC *non-blocking* [8, 22]. To give better performance, the vote and decision are sent in a ring instead of back and forth between the primary and backup. The protocol is always single-fault tolerant and these methods could be favorably applied in a shared-nothing, fault-tolerant DBMS like ClustRa [13].

The rest of the paper is organized as follows: Section II.2 summarizes related work. Section II.3 presents the system model and Section II.4 defines the non-blocking atomic commitment problem. Section II.5 gives an overview and a detailed description of C2PC, proves the correctness of the protocol, and outlines a one-phased version called C1PC. Then, an evaluation of the protocols is given in Section II.6. Finally, the conclusion and further work are presented in Section II.7.

## II.2 Related Work

Several atomic commitment protocols and variations have been proposed over the years. Many approaches have been concerned with either developing a non-blocking protocol or the performance issues. However, only a few deal with both.

In a non-replicated environment, 2PC may block if the coordinator and a participant fail [8, 3]. 3PC [26] decreases the chance of blocking failures by adding an extra round of messages, thus favoring resilience over performance.

3PC has been extended to partitioned environments [21], and the number of communication steps has been reduced to the same as 2PC by using consensus [9], causing an increase in the number of messages or requiring broadcast capabilities.

Several 2PC-based modifications where performance issues are handled exist [24]. Presumed commit and presumed abort [19] both avoid one flushed disk write, by assuming that a non-existent log record means that the transaction has committed or aborted, respectively. Transfer-of-commit, lazy commit and read-only commit [8], sharing the log [19, 28] and group commit [6, 20] are other optimizations. An optimization of the presumed commit protocol [15] reduces the number of messages, but requires the same number of forced disk writes.

Optimistic commit protocols are designed to give better response time during normal processing, but will need extra recovery after failures or aborts. They release locks when the transaction is prepared, but must be able to handle cascading aborts by using semantic knowledge [17]. PROMPT [12] uses optimistic locking in the sense that locks can be lent to other transactions after the participant has voted yes. A transaction that lends locks will not reply to the request until the locks are fully released by the previous transaction, and only one transaction at a time can lend a lock. This approach avoids cascading aborts while it may yield better performance because of increased concurrency.

One-phased commit protocols have also been proposed [28, 2, 1, 16, 29]. These are based on the early prepare or unsolicited vote method by Stonebraker [30] where the prepare message is piggybacked on the last operation sent to a participant. In this way, the voting phase is eliminated. However, these approaches inflict strong assumptions and restrictions on the transactional system [1]. For instance, it requires either the participants to prepare the transaction for each request-reply interaction, or the coordinator must be able to identify the last request for a transaction to be able to piggyback a prepare-request. Otherwise, the performance of 1PC degrades.

A few approaches that render 2PC non-blocking by replication have been proposed. The first replicates the coordinator, but not the participants [22]. In addition to sending log records to the backup, they are forced to disk, causing a decrease in performance. Also, the backup only finishes transactions already started. No new transactions can be initiated by the backup. This approach has also been adapted to multiple backups [23].

The second combines optimistic commit and replication [14]. A replicated group of commit servers is used to keep the log records not yet written to the log by the participant available, thus ensuring resilience to failures. This approach uses multicast and has the same latency as 2PC, but requires more

messages to be sent.

A third approach [18] is the most similar to the approach adopted in this paper. The differences are that it incurs unnecessary overhead by sending the “start of prepare” and the commit log records to the backup, and it forces log records to the disk even if both the primary and the backup work correctly. The performance is thus degraded.

### II.3 System Model

The system is composed of a number of processes or nodes connected through a communication network. Each process has both a functional unit (application or database server) and a transaction manager. A process executes two kinds of actions. (1) Change state and (2) send or receive a message. When correct, they execute at arbitrary speeds, but eventually make progress. Processes fail by crashing, causing them to lose state. Such events are, however, rare. A failed process is recovered and brought up-to-date by the system.

Communication is *asynchronous* and *reliable*. Thus, there are no bounds on communication delays and messages are not corrupted or lost if both the receiving and the sending process behaves correctly, i.e. do not crash.

In an asynchronous system, a failure detector is needed to make the system reliable [5]. An *eventually strong* failure detector can solve the atomic commitment problem [10]. However, to simplify the problem descriptions and explanations, a perfect failure detector that eventually suspects every faulty process and never suspects a correct process is assumed.

For the purpose of this paper, no disks are used. State is stored entirely in main memory. Thus, to make the state persistent and the system highly available the *primary-backup approach* [4] is used. This approach assumes that MTTF is orders of magnitude larger than MTTR, thus both the primary and backup do not fail at the same time.

Following [19], the costs of execution in this system are twofold. (1) The computation cost is the total number of messages sent, and (2) the delay is serialized messages. The main-memory operations associated with the atomic commitment protocol are only a small fraction of the load on the system, thus their costs are assumed to be negligible. Also, as long as the cpus are not fully utilized, there are no queueing effects.

## II.4 The Non-Blocking Atomic Commitment Problem

An atomic commitment protocol ensures that the participants in a transaction agree on the outcome, i.e. ABORT or COMMIT. Each participant votes, YES or NO, on whether it can guarantee the local ACID properties of the transaction. All participants have a right to *veto* the transaction, thus causing it to abort. The *Non-Blocking Atomic Commitment* problem, NB-AC, has these properties [3, 10]:

**NB-AC1** *<uniform agreement>* All processes that decide reach the same decision.

**NB-AC2** *<integrity>* A process cannot reverse its decision after it has reached one.

**NB-AC3** *<uniform validity>* COMMIT can only be reached if *all* processes voted YES.

**NB-AC4** *<non-triviality>* If there are no failures and no processes voted NO, then the decision will be to COMMIT.

**NB-AC5** *<termination>* Every correct process eventually decides.

## II.5 The Circular Two-Phase Commit Protocol

This section presents the *Circular Two-Phase Commit protocol*, C2PC, for main-memory primary-backup systems.

Normally, 2PC requires both forced and non-forced disk writes [19, 8]. In a primary-backup environment, these disk writes can be replaced by, respectively, synchronous (blocking) and asynchronous (non-blocking) logging to the backup node. Figure II.1(a) illustrates this. The small arrows between each primary-backup pair is the logging.

2PC (Figure II.1(a)) consists of two phases, a voting phase and a decision phase. In the voting phase, the votes are collected by a coordinator, and the coordinator makes a decision depending on the votes and persistently stores the decision. In the decision phase, the outcome is sent to the participants which send an acknowledgement back to the coordinator. Each participant

must persistently store its vote and the outcome before replying to the coordinator in, respectively, the voting and decision phase. After the decision has been made persistent, the coordinator can give an *early answer* [13] to the client. Thus, the response time seen from the client is less than what it would be if the second phase had to be completed before the reply.

1PC (Figure II.1(b)) piggybacks the prepare-message on the last work request for the transaction. Thus, the first phase of the voting is eliminated. However, each participant's vote must be persistently stored to the backups before replying to the coordinator.

The C2PC protocol is a modified version of 2PC for main-memory primary-backup systems. Similarly to 2PC, C2PC has two phases and logs the votes

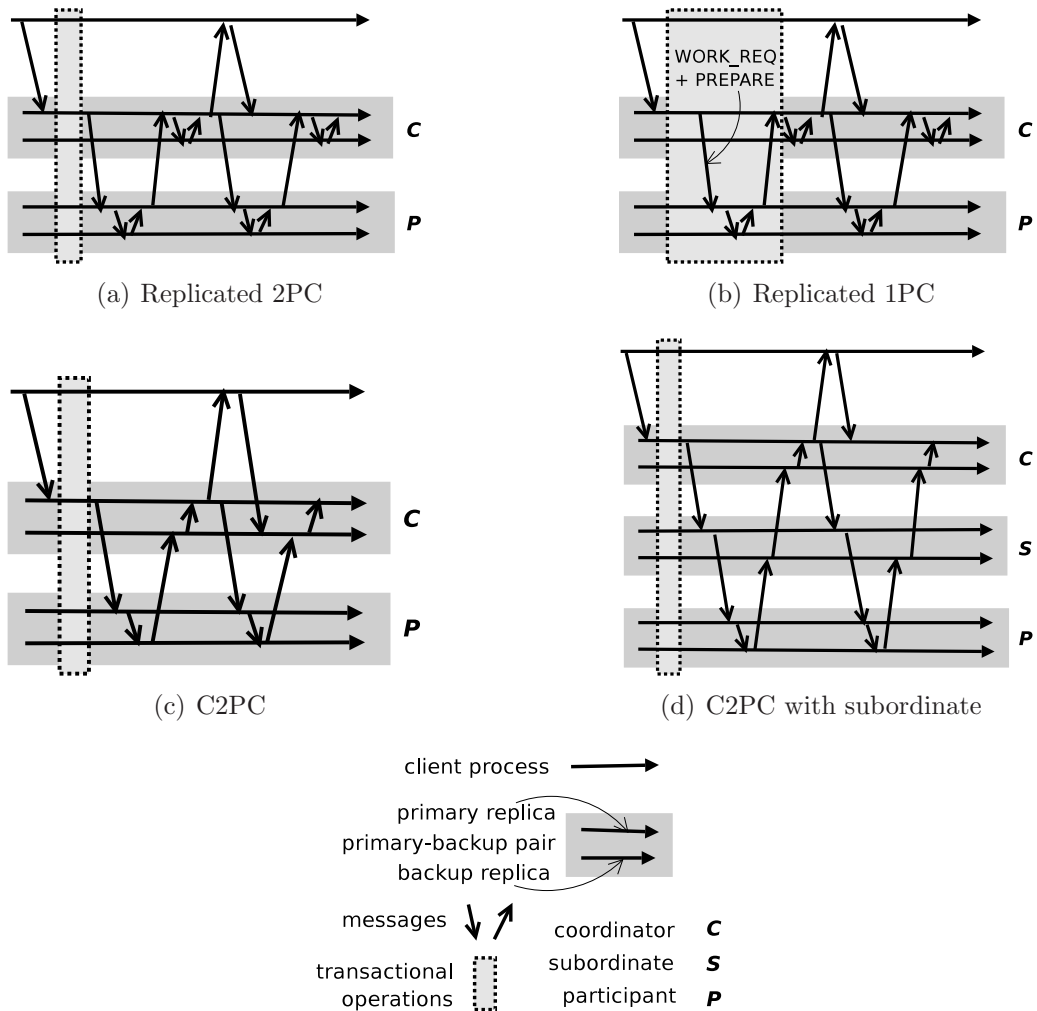


Figure II.1: Execution of various atomic commitment protocols

and decision to the backups. However, it allows the backup to reply to the backup coordinator. This is shown in Figure II.1(c). Instead of sending votes and acknowledgments back and forth, the votes and decision are sent in a ring for each branch of the commit tree. This is a case of the transfer-of-commit optimization [8] where the authority to commit is passed via the participants to the backup root coordinator.

C2PC reduces both the number of messages in the critical path and the total number of messages to commit a transaction. The critical path is the delay until the transaction coordinator can give an early answer to the client. For instance, comparing Figure II.1(a) and II.1(c), the added delay has been reduced from six to four messages and the added number of messages from thirteen to nine. By comparison, 1PC (Figure II.1(b)) has an added delay of four, two within the transactional operations frame and two after, and a total overhead of eleven messages.

During normal processing, the communication goes through each ring twice, one for each phase as seen in Figure II.1(c). In the first round, the primary coordinator, *pc*, votes and piggybacks its own vote on the prepare message to the primary participant, *pp*. Each *pp* votes and sends its vote along with the vote of the *pc* to the backup participant, *bp*. *Bp* adds its own vote and forwards it to the backup coordinator, *bc*. *Bc* makes a decision based on the received votes and its own. The decision is then made persistent by sending it to the *pc*, which gives an early answer to the client and initiates the second phase.

The protocol also handles subcoordinators, or *subordinate* processes [19]. A subordinate acts as a participant to the coordinator and as a coordinator to the participants. A subordinate can also act as a participant to another subordinate. During the first phase a primary subordinate, *ps*, votes and forwards the vote to each of the subparticipants. The backup subordinate, *bs*, collects the votes from all the subparticipants before it sends its vote to the *bc*. During the second phase, the decision is propagated in the same fashion.

If, during the first phase, one of the participants or subordinates votes NO, the vote is propagated back to the *bc*, while each subordinate along the way makes the decision to abort. The decision is then sent out to all remaining undecided participants and subordinates.

The protocol handles failures of both the primary and the backup. These failure scenarios might occur:

- If one of the primaries fails during the first phase, the transaction is aborted as the backup cannot be sure that it has all the log records.
- If one of the backups fails during the first phase, the preceding node in

the ring sends the vote message to the primary instead.

- If one of the participating primaries (resp. backups) fails during the second phase, the preceding node in the ring sends the decision or acknowledgement message to the backup (resp. primary) instead.

Rerouting the messages to the non-failed primary or backup in the last two scenarios above works since the primary and backup is assumed never to fail at the same time.

First, a detailed explanation is given, second, the correctness of the protocol is proven and, third, a one-phase version of C2PC, C1PC, is outlined.

### II.5.1 Detailed Description

This section presents the C2PC protocol in detail. Listings II.1 to II.6 present the protocol in failure-free scenarios for all types of nodes.

Each process has a Transaction Table (TT) which holds the state (*active*, *prepared*, *committed*, or *aborted*) and known participants of each transaction. Also, it is told which processes have failed from the local failure detector. Log records marked with a Log Sequence Number (LSN) [3] are shipped asynchronously to the backup. The backup checks that it has received all LSNs and acknowledges the greatest LSN received so far. The TT of the primary holds the greatest LSN acknowledged so far by the backup, and the backup TT is updated as log records are received and acknowledged from the primary. When voting, any unacknowledged log records are piggybacked on a `VoteMsg`. The TT can also be changed by receiving a vote message, `VoteMsg`, from a participant.

First, the protocols for the coordinator and the participants are presented. Then, the protocols for the subordinates are given.

#### Coordinator and Participants

As seen in Listing II.1, the *pc* of the transaction initiates the protocol by attaching its own vote to a `VoteMsg` and sending it to each of the participants.

Some necessary information is included in all messages going down in the commit tree: (1) The transaction identifier, (2) the address of the primary and backup of the *pc* and (3) the address of the client. The first identifies the transaction to be committed, while the second allows *bp* to contact *bc*. The third allows *bc* to contact the client to complete the transaction in case *pc* should fail. Also, included in at least one of the vote messages are (4) the unacknowledged log records of the transaction at *pc* and (5) a list of the

```

atomic_commitment:                                1
if (myVote == NO) {
    decide(ABORT);
    voteMsg = new VoteMsg(txn,No);                4
    send (voteMsg) to all participants;
} else {
    voteMsg = new VoteMsg(txn,Yes);                7
    send (voteMsg) to all participants;
    receive(DecisionMsg) from backup {
        if (decision is COMMIT) decide(COMMIT);    10
        else decide(ABORT);
        dMsg = new DecisionMsg(txn,decision);
        send reply to client;                       13
        send (dMsg) to all participants;
        if (decision is COMMIT) {
            receive (AckMsg) from backup;           16
            on timeout {resend dMsg;}
        }
    }
} }

```

Listing II.1: Primary coordinator

```

atomic_commitment:
receive (voteMsg) from all participants;           20
if (receivedVotes == NO || myVote == NO) {
    decide(ABORT);
    dMsg = new DecisionMsg(txn,ABORT);             23
} else {
    decide(COMMIT);
    dMsg = new DecisionMsg(txn,COMMIT);           26
}
send (dMsg) to primary;
if (decision is COMMIT) {                          29
    receive (DecisionMsg) from all;
    receive ack from client;
    send (AckMsg) to primary;                       32
}

```

Listing II.2: Backup coordinator

participants of the transaction. The fourth ensures that *bc* has all the log records generated by *pc* of the transaction before committing it. Finally, the fifth guarantees that *bc* waits for `VoteMsgs` from all the participants before making a decision and enables it to complete a transaction in case *pc* fails.

Each *pp* (Listing II.3) of the transaction receives a `VoteMsg`. If the received vote or its own is NO, the decision is ABORT, and a new `VoteMsg` with a NO-vote is sent to the backup. If the vote is YES, *pp* adds its unacknowledged log records for the transaction to the vote message and forwards it to the backup.

When a YES-vote is received by a *bp* (Listing II.4), the log records from

```

atomic commitment:
receive(voteMsg);
if (receivedVote == NO || myVote == NO) {
    decide(ABORT);
    vMsg = new VoteMsg(txn,NO);
    send (vMsg) to backup;
} else {
    txnLog = getLog(txn);
    vMsg = new VoteMsg(txn,vote,info);
    send (vMsg) to backup;
    receive(decisionMsg) {
        decide(decisionMsg.decision);
        send (decisionMsg) to backup;
    }
}

```

Listing II.3: Primary participant

```

atomic_commitment:
receive (voteMsg);
if (receivedVote == NO || myVote == NO) {
    decide(ABORT);
    vMsg = new VoteMsg(txn,NO);
    send (vMsg) to parentBackup;
} else {
    vMsg = new VoteMsg(txn,YES);
    send (vMsg) to parentBackup;
    receive (decisionMsg) {
        decide(decisionMsg.decision);
        send (AckMsg) to parentBackup;
    }
}

```

Listing II.4: Backup participant

the *pp* are removed from the `VoteMsg` and applied to the local log. The local vote is then collected and forwarded to the backup of the parent. If the local vote or the received one is NO, the decision is ABORT and a `VoteMsg` containing a No-vote is forwarded to *bc*.

Listing II.2 shows the algorithm for *bc*. Upon receiving a `VoteMsg`, it checks if the message contains a list of participants. If so, it checks whether or not it has received a `VoteMsg` from all of them. If a list is not included, it knows that there are more messages coming. Either way, it waits until all participants' votes have been collected (line 20), and then makes a decision: ABORT if any NO-votes have arrived or itself votes NO, otherwise COMMIT. Any unacknowledged log records sent from *pc* are appended to the local log and a decision message, `DMsg`, is then sent to *pc*.

When the *pc* receives a `DMsg`, it decides the same and then forwards the decision to the client and the participants. If the decision is COMMIT, it waits to receive a confirmation from *bc* saying that all participants have committed

before the transaction can be removed from TT.

After voting, the participants wait for a decision. When received, the decision is made, and the message is sent from the *pp* to the *bp* to the *bc*. Note that, since the *pc* and the *bc* are assumed not to fail at the same time, a termination protocol is not needed for the participants, because the coordinator ensures the liveness of the transaction.

### Subordinate processes

The previous subsection is necessary to make an atomic commitment, but internal nodes in the commit tree can also exist. These nodes are called *subordinates* [19] and are characterized by acting as a coordinator for some participants, while being a participant itself for the coordinator or other subordinate.

The protocol for a primary subordinate, *ps*, is given in Listing II.5. When a `VoteMsg` is received, it decides `ABORT` if the received or its own vote is `NO`. Otherwise, the unacknowledged log records are appended to at least one of the outgoing `VoteMsgs` along with a list of the participants. Either way, the address of the current *ps* and *bs* is sent to the participants along with the vote and the information received in the `VoteMsg`.

A backup subordinate, *bs*, (Listing II.6) waits, as the *bc*, until a `VoteMsg` is received from all its participants and then makes a decision based on the received votes and, if all votes are `YES`, the result of applying the log records received from the primary. The information from the parent primary is added to the `VoteMsg`, and it is sent to the parent backup.

In the same way as the participants, the subordinates wait for a decision after voting. When received, the decision is made, and the message is sent from the *ps* to a *pp* or another *ps*. The *bs* receives the decision from one or more *bps* or *bss* and forwards it to the *bc*. For the same reasons as for the participants, a termination protocol is not needed here.

### II.5.2 Correctness

This section proves the correctness of the C2PC protocol by proving each of the properties given in Section II.4 in this order: **NB-AC2**, **NB-AC3**, **NB-AC4**, **NB-AC1** and **NB-AC5**.

**Lemma 1. NB-AC2:** *A process cannot reverse its decision after it has reached one.*

*Proof.* The algorithms for each of the processes use if-else statements to avoid deciding more than once per process. ■

```

atomic_commitment:                                     59
receive(voteMsg);
if (receivedVote == NO || myVote == NO){
    decide(ABORT);                                     62
    vMsg = new VoteMsg(txn,NO);
    send (vMsg) to participants;
} else {                                             65
    txnLog = getLog(txn);
    voteMsg = new VoteMsg(txn,vote);
    send (voteMsg) to participants;                   68
    receive(decisionMsg) {
        decide(decisionMsg.decision);
        send (decisionMsg) to participants;           71
    }
} }

```

Listing II.5: Primary subordinate

```

atomic_commitment:                                     74
receive (voteMsg) from all subparticipants;
if (receivedVotes == NO || myVote == NO) {
    decide(ABORT);
    vMsg = new VoteMsg(txn,NO);                         77
    send (vMsg) to parentBackup;
} else {
    vMsg = new VoteMsg(txn,YES);                         80
    send (vMsg) to parentBackup;
    receive (decisionMsg) {
        decide(decisionMsg.decision);                   83
        send (DecisionMsg) to parentBackup;
    }
} }

```

Listing II.6: Backup subordinate

**Lemma 2. NB-AC3:** *The COMMIT decision can only be reached if all processes voted YES.*

*Proof.* All processes can decide COMMIT during the second phase of the protocol. However, they can only decide COMMIT if they receive a message with a COMMIT decision. The only process that can decide COMMIT during the first phase is *bc* (line 25). This happens only if it has received YES-votes from all the participating processes including itself. ■

**Lemma 3. NB-AC4:** *If no process failed and no process voted NO, then the decision will be COMMIT.*

*Proof.* If no process failed and no process voted NO, then, since the communication system is reliable, *bc* receives YES from all participants and subordinates. Thus, COMMIT is reached (line 25). ■

**Lemma 4. NB-AC1:** *All processes that decide reach the same decision.*

*Proof.* A process can only decide ABORT during the second phase, if a process decided ABORT during the first phase. Similarly, a process can only decide COMMIT during the second phase, if  $bc$  decided COMMIT during the first phase. As proved in Lemma 2, COMMIT can be decided (line 25) only if all processes voted YES. A process can only decide ABORT during the first phase if it votes NO. A process cannot both vote YES and NO, so two processes cannot decide differently. ■

**Lemma 5. NB-AC5:** *Every correct process eventually decides.*

*Proof.* To enable a process to decide in the presence of failures, all failure scenarios as well as the scenario with no failures must be handled. These scenarios can occur:

- 1:  $Pc$  fails before sending the vote to all participants.
- 2:  $Pc$  fails after initiating the voting, but before sending the decision to all participants.
- 3:  $Pc$  fails after sending the decision to all participants, but before receiving an AckMsg from  $bc$ .
- 4:  $Bc$  fails before sending the decision to  $pc$ .
- 5:  $Bc$  fails after sending the decision to  $pc$ , but before sending AckMsg to  $pc$ .
- 6: A  $ps$ ,  $bs$ ,  $pp$ , or  $bp$  fails before sending the vote.
- 7: A  $ps$ ,  $bs$ ,  $pp$ , or  $bp$  fails after sending the vote, but before sending the decision.
- 8: No node fails.

Scenario (1): None has voted, each of the participants can independently abort the transaction after a timeout has expired without causing inconsistencies in the system.

Scenario (2): When  $bc$  does not receive a decision from any of the participants and  $pc$  fails,  $bc$  can complete the transaction with the decided outcome.

Scenario (3): The AckMsg is sent to  $pc$  to allow it to purge the transaction entry from its TT. However, this is not needed if  $pc$  fails, because it will have to update its TT as part of the recovery process.

Scenarios (4) and (6): If  $pc$  does not receive a decision within a given time limit, it can send a message to  $bc$  and tell it about the timeout, then  $bc$  can decide Abort. If  $bc$  has failed,  $pc$  can safely abort the transaction.

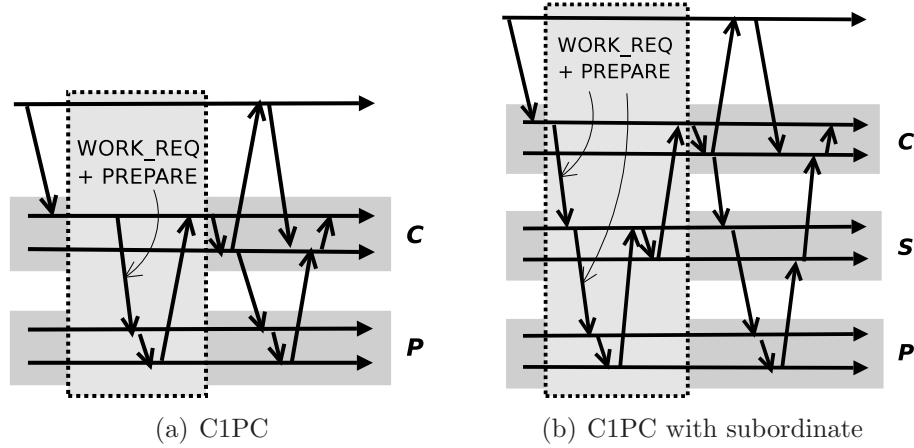


Figure II.2: Examples of C1PC execution

Scenario (5): The transaction is completed, but if the decision is COMMIT the transaction entry will not be deleted from the TT of  $pc$  until an `AckMsg` is received. However,  $pc$  resends the decision (line 17) with updated backup information until it receives confirmation that all participants have decided.

Scenario (7): When a process fails during the second phase, the decision must be sent via the backup on its way down the commit tree or via the primary on its way up the tree.  $Pc$  resends the decision (line 17) until it receives an acknowledgement, and the failed processes are bypassed.

Scenario (8): This is proven similarly to Lemma 3. Since no process failed and the communication system is reliable,  $bc$  receives votes from all participants and subordinates. Thus, it decides either COMMIT in line 25 or ABORT in line 22. By the same argument, each participant and subordinate eventually decides.

All scenarios are handled, thus, all correct processes eventually decide. ■

**Theorem 1.** *C2PC is a valid non-blocking atomic commitment protocol.*

*Proof.* Since C2PC satisfies properties NB-AC1 - NB-AC5 it solves NB-AC. ■

### II.5.3 C1PC

*Circular One-Phase Commit protocol*, C1PC, is a circular version of 1PC and can be designed as shown in Figure II.2. The main differences between C1PC and C2PC are: During the first phase (1)  $pc$  piggybacks `VoteMsg` on the last request and (2)  $bp$  replies to  $ps$  or  $pc$  (instead of  $bs$  or  $bc$ ) because there might be results that are needed. During the second phase, (3)  $pc$  makes the

Protocol	Delay	Total
Non-fault tolerant	0	0
Replicated 2PC, parallel	6	$8N + 5$
Replicated 2PC, linear	$4N + 4$	$8N + 5$
Replicated 1PC, parallel	4	$6N + 5$
Replicated 1PC, linear	$2N + 2$	$6N + 5$
C2PC, parallel	4	$8N + 1$
C2PC, linear	$2N + 1$	$4N + 5$
C2PC, hybrid	4	$6N + 1$
C1PC, parallel	2	$4N + 3$
C1PC, linear	$N + 1$	$3N + 4$
C1PC, hybrid	2	$3N + 4$

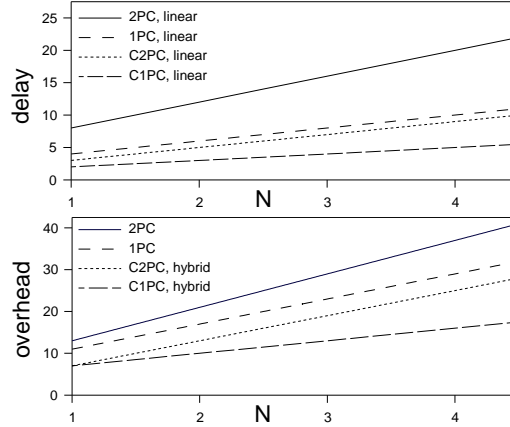


Figure II.3: Added delay until early answer to client and total overhead for various ACPs.  $N = \#$  servers invoked by transaction excluding the coordinator,  $N \geq 1$

decision to commit, and (4) *bc* replies to the client and sends the **DMsg** to the participants.

## II.6 Evaluation

This section compares the performance of non-fault tolerant, replicated 2PC, replicated 1PC, C2PC and C1PC. We assume the normal operational mode where no participating process fails and all participants vote YES. The purpose is to evaluate the costs associated with the various protocols.

The table in Figure II.3 shows formulas for the added number of messages in the critical path and the total overhead to complete a transaction compared to the non-fault-tolerant case. The critical path is the delay until the transaction coordinator can give an early answer to the client. Parallel and linear execution corresponds to a commit-tree of height 1 and  $N - 1$ , respectively.

The non-fault-tolerant case is non-replicated and has zero delay and overhead to complete the request. It does not tolerate any failures and there is no coordination of the outcome.

For the transactional cases, the parallel versions of C2PC and C1PC have the shortest delay and the linear versions have the least overhead. This observation leads to the *hybrid* versions of C2PC and C1PC, where the voting

phase is executed in parallel and the decision phase in linear. This minimize both the delay and the overhead.

The graphs in Figure II.3 depicts the delay and overhead of selected protocols. The protocols with constant delay are not shown in the delay graph and in the overhead graph the linear and parallel circular protocols are not showed to avoid cluttering.

The delay of parallel and hybrid C2PC is equal to and two-thirds of the delay of 1PC and 2PC, respectively. C1PC halves the delay and almost halves the overhead compared to 1PC, but also inherits its restrictions and assumptions [1]. The overhead of the parallel and hybrid versions of C1PC is almost half of that of 1PC, and hybrid C2PC has less overhead than 1PC.

## II.7 Conclusion

This paper has presented an atomic commitment protocol, Circular Two-Phase Commit (C2PC). It is a single-fault-tolerant optimization of 2PC for replicated main-memory primary-backup systems. C2PC does not require any changes to the standard 2PC interface and can be implemented in an asynchronous system with an unreliable failure detector. The protocol is unique in the sense that it does not log to disk and ensures liveness for both data, processing and transaction commitment.

For further work the protocol should be implemented and performance measures should be made to verify the analysis and evaluation in Section II.6.

## References

- [1] Maha Abdallah, Rachid Guerraoui, and Philippe 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] Maha Abdallah and Philippe Pucheral. A single-phase non-blocking atomic commitment protocol. In *DEXA '98: Proceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 584–595, London, UK, 1998. Springer-Verlag.
- [3] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publ. Co., Inc., 1987.

- 
- [4] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems. In S. Mullender, editor, *Distributed Systems*, ACM Press, chapter 8: The primary-backup approach, pages 199–216. Addison-Wesley, second edition, 1993.
  - [5] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
  - [6] Dieter Gawlick and David Kinkade. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985.
  - [7] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.
  - [8] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
  - [9] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 692–697, Hong Kong, 1996.
  - [10] Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972, pages 87–100, Le Mont-Saint-Michel, France, 1995. Springer-Verlag.
  - [11] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.
  - [12] Jayant R. Haritsa, Krithi Ramamritham, and Ramesh Gupta. The PROMPT real-time commit protocol. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):160–181, 2000.
  - [13] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, 1995.

- 
- [14] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Sergio Arévalo. A low-latency non-blocking commit service. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 93–107. Springer-Verlag, 2001.
- [15] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th Conference on Very Large Databases*. Morgan Kaufman, 1993.
- [16] Inseon Lee and Heon Young Yeom. A single phase distributed commit protocol for main memory database systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 44–51, Washington, DC, USA, 2002. IEEE Computer Society.
- [17] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. An optimistic commit protocol for distributed transaction management. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, pages 88–97, New York, NY, USA, 1991. ACM Press.
- [18] Sharad Mehrotra, Kexiang Hu, and Simon Kaplan. Dealing with partial failures in multiple processor primary-backup systems. In *CIKM '97: Proceedings of the Sixth International Conference on Information and Knowledge Management*, pages 371–378, New York, NY, USA, 1997. ACM Press.
- [19] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R\* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.
- [20] Taesoon Park and Heon Y. Yeom. A consistent group commit protocol for distributed database systems. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS'99)*, 1999.
- [21] Michael Rabinovich and Edward D. Lazowska. A fault-tolerant commit protocol for replicated databases. In *PODS '92: Proceedings of the eleventh ACM SIGMOD symposium on Principles of Database Systems*, pages 139–148, New York, NY, USA, 1992. ACM Press.
- [22] P. Krishna Reddy and Masaru Kitsuregawa. Reducing the blocking in two-phase commit protocol employing backup sites. In *Conference on Cooperative Information Systems*, pages 406–416, 1998.

- 
- [23] P. Krishna Reddy and Masaru Kitsuregawa. Blocking reduction in two-phase commit protocol with multiple backup sites. In *DNIS '00: Proceedings of the International Workshop on Databases in Networked Information Systems*, pages 200–215, London, UK, 2000. Springer-Verlag.
- [24] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 520–529, Washington, DC, USA, 1993. IEEE Computer Society.
- [25] Fred B. Schneider. Replication management using the state machine approach. In *Distributed systems (2nd Ed.)*, pages 169–197. ACM Press/Addison-Wesley Publishing Co., 1993.
- [26] Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142. ACM Press, 1981.
- [27] Peter M. Spiro, Ashok M. Joshi, and T. K. Rengarajan. Designing an optimized transaction commit protocol. *Digital Technical Journal*, 3(1):70–78, 1991.
- [28] James W. Stamos and Flaviu Cristian. A low-cost atomic commit protocol. In *Proceedings of the Ninth Symposium of Reliable Distributed Systems*, 1990.
- [29] James W. Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4), 1993.
- [30] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.



# Paper III

Efficient High-Availability Commit Processing.

Heine Kolltveit and Svein-Olaf Hvasshovd.

*In Proceedings of the Third International Conference on  
Availability, Reliability and Security, IEEE*

March 4 - 7, 2007, Barcelona, Spain. *Shortlisted for Best Paper  
Award (1 of 4).*



# Efficient High-Availability Commit Processing

Heine Kolltveit and Svein-Olaf Hvasshovd

Department of Computer and Information Science  
Norwegian University of Science and Technology  
NO-7491 Trondheim, Norway

## Abstract

Distributed transaction systems require an atomic commitment protocol to preserve ACID properties. A commit protocol should add as little overhead as possible to avoid hampering performance. In this paper, dynamic coordinators are introduced. In main-memory primary-backup systems, the approach significantly reduces the time spent during commit processing. The performance of such protocols must be properly evaluated to give system developers the information needed to make an educated choice between them. Thus, simulation results, verified by statistical analysis, are presented. The simulation results show that the performance can be significantly boosted by using optimizations and protocols especially designed for high-availability main-memory systems.

## III.1 Introduction

Computer systems are getting increasingly more complex, which puts increasing pressure on the performance of each of the components. For databases, being a central building block, it is crucial that they are fast and reliable. Declining main-memory (MM) prices and increasing storage capacity have contributed to the advent of MM databases and applications. Contrary to traditional disk-resident databases, main-memory databases (MMDB) avoid slow disk accesses by *permanently* storing all data in main memory [9]. This speeds up data access, since access to main memory is much faster than access to disk [8, 9].

MMDBs can be classified according to the *log policy* and their *persistence policy*. The log policy determines if the log is saved to disk, or is only

kept in main memory. The persistence policy determines if the log is persistently stored or not. When logging to disk, the log is persistently stored if *synchronous logging* is used. If not, the log cannot be guaranteed to be recoverable. In *pure* MM systems, neither data nor log resides on disk. The log and data can be made persistent by *replicating* it between processes at separate nodes. This paper is in the context of persistent pure main memory databases.

Messages are sent to replicate data between processes. Rather than the long disk latency for disk-based systems, a shorter network latency is introduced. Thus, the performance of a MM system has the possibility of outperforming a disk-based solution. Also, since there exist multiple copies of the data, a crash failure [7] will only bring down one of the copies. Thus, high data availability is achieved.

Tailored protocols are needed to enable MM systems to exploit the potential performance advantages of their nature. A transactional system, such as a database, requires an *Atomic Commitment Protocol* (ACP) to ensure ACID properties [12]. Many ACPs have been designed [19, 25], but although performance evaluations by simulating commit protocols have been done [6, 42, 5, 14, 32, 15, 36, 24, 25], none are targeted at pure MM protocols.

This paper is motivated by the observation that transactions are often sent to a dedicated transaction coordinator node, which coordinates the transaction execution and commit processing. This can cause a bottleneck. A solution is to balance the load and allow all nodes to be transaction coordinators. However, if the transaction coordinator is not a participant of the transaction, it results in unnecessary overhead. Fewer nodes involved in commit processing means less overhead. In this paper, *dynamic coordinators* are used to enhance the performance of MM commit protocols. In addition, there is a lack of simulation and statistical analysis for MM commit protocols. Without a performance evaluation, the protocols' effectiveness have not been sufficiently verified. Since there was no large computing cluster available for testing purposes, simulation was chosen as the method of evaluation.

The main contributions of this paper are novel optimizations for MM commit protocols and analytically verified simulation results comparing the relative performance of MM commit protocols with and without these optimizations. Also, a framework for analysing the performance impact for various MM commit protocols and enhancements is outlined. The results of such an evaluation can favorably be used by system developers to improve the performance of MM applications, e.g., in a shared-nothing fault-tolerant DBMS like ClustRa [16].

This paper only simulates failure-free execution where all transactions commit. Since this is the dominant execution path, it is where the potential

for performance gains is greatest.

The rest of the paper is organized as follows. Section III.2 gives an overview of related work. Section III.3 presents the commit protocols discussed in this paper, while Section III.4 introduces the new optimizations using dynamic coordinators. Section III.5 outlines the simulation model and parameters used for the simulations in Section III.6. The simulations are compared to a statistical analysis in Section III.7, while Section III.8 gives some concluding remarks.

## III.2 Related Work

### III.2.1 Optimizations

Several 2PC-based modifications exist where performance issues are handled [35]. Presumed commit and presumed abort [29] both avoid one flushed disk write, by assuming that a non-existent log record means that the transaction has committed or aborted, respectively. Transfer-of-commit, lazy commit, and read-only commit [13], sharing the log [29, 38], and group commit [10, 34] are other optimizations. An optimization of the presumed commit protocol [21] reduces the number of messages, but requires the same number of forced disk writes.

Optimistic commit protocols are designed to give better response time during normal processing, but will need extra recovery after failures or aborts. They release locks when the transaction is prepared, but must be able to handle cascading aborts by using semantic knowledge [26]. PROMPT [15] uses optimistic locking in the sense that locks can be lent to other transactions after the participant has voted yes. A transaction that lends locks will not reply to the request until the locks are fully released by the previous transaction, and only one transaction at a time can lend a lock. This approach avoids cascading aborts while it may yield better performance because of increased concurrency.

One-phased commit protocols have also been proposed [38, 2, 1, 24, 39, 16]. These are based on the early-prepare or unsolicited-vote method by Stonebraker [40] where the prepare message is piggybacked on the last operation sent to a participant. In this way, the voting phase is eliminated. However, these approaches may inflict strong assumptions and restrictions on the transactional system [1].

One existing approach combines optimistic commit and replication [18]. A replicated group of commit servers is used to keep the log records not yet written to the log by the participant available, thus ensuring resilience to

failures. This approach uses multicast and has the same latency as 2PC, but requires more messages to be sent.

Commit protocols for replicated main-memory systems have been introduced by Kolltveit and Hvasshovd [19] and in the context of ClustRa [16]. These are presented in greater detail in Section III.3.

### III.2.2 Performance evaluations

Performance evaluations are presented alongside most proposed commit protocols. For most of them, this involves nothing more than counting messages and log writes required to terminate a transaction, as initially done for 2PC in [11].

Even if simulation results comparing different types of commit processing exist for some cases, most of them assume logging to disk. The impact of communication and site failures for 2PC, optimistic 2PC, and a version which resends requests before giving up are presented by Boutros and Desai [6]. Also, Liu, Agrawal, and Abbadi [27] simulate 2PC, presumed commit, presumed abort, and early prepare in the presence of site failures.

Simulations comparing 2PC, 3PC, presumed abort, presumed commit, and an optimistic protocol as well as two baseline protocols, one for a completely centralized system, and the other for distributed transaction processing and centralized commit processing, have been performed [14]. The study considers the effects of resource and data contention, the multiprogramming level, slow and fast network interfaces, the degree of distribution, and aborts. Simulations of the same protocols in hard real-time environments, where a transaction is aborted if it is not terminated within a given timeframe, have also been done [15].

In a study presented by Samaras, Georg, and Chrysanthis [36], simulation results show that restructuring of the commit tree may yield better response time.

A simulation of commit protocols in gigabit networks [42] compares the performance of the three types of standard two-phase commit (basic, presumed abort, and presumed commit [29]) and two one-phased protocols (Coordinator Log [39] and Implicit Yes-Vote (IYV) [4]). The study shows that the one-phased protocols, with IYV at the top, are better, when applicable, for high-speed networks.

A simulation study for main-memory commit protocols exist [24], which is further refined in [23, 25]. However, this approach assumes synchronous logging to disk, severely degrading the commit processing compared to a pure main-memory approach.

Many statistical queuing analyses of transactional systems have been performed, and several surveys of them have been conducted [3, 41, 31]. They are mostly focused on other aspects of transaction processing, mainly concurrency control, and the statistical analysis of different types of commit processing have been overlooked.

## III.3 Commit Protocols

This section presents commit protocols for main-memory primary-backup systems. They are introduced by outlining the standard commit protocols that have influenced them.

### III.3.1 Two-Phase Commit

The Two-Phase Commit Protocol (2PC) [11, 30] consists of two phases: The *voting phase* and the *decision phase*. During the voting phase the votes are collected from the transaction participants by the transaction coordinator: It sends out a **Prepare** message. Then, before replying with a **Vote** message, each participant makes its vote *durable* by forcing to stable storage [22]. The coordinator makes a decision based on the votes. A missing or negative vote is interpreted as a veto, and it decides to **ABORT** the transaction. Otherwise, if the coordinator agrees with a positive outcome, the decision is **COMMIT**. The decision is then forced to stable storage.

In the decision phase, the decision is propagated to the participants, which force the decision to stable storage and reply **Committed** to the coordinator and forget the transaction. After receiving **Committed** from all participants, the coordinator writes a committed log record and forgets about the transaction. This record does not need to be forced.

### III.3.2 Replicated Two-Phase Commit

The *Replicated Two-Phase Commit Protocol* (R2PC) is a simple modification of 2PC to a main-memory primary-backup environment. In such an environment the forced and non-forced disk writes can be replaced by, respectively, synchronous (blocking) and asynchronous (non-blocking) logging to the backup node. Fig. III.2(a) illustrates this. The small arrows between each primary-backup pair is the logging. The rest of the protocol remains unchanged.

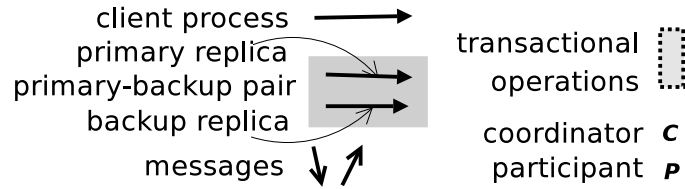


Figure III.1: Legend for Fig. III.2 and Fig. III.3

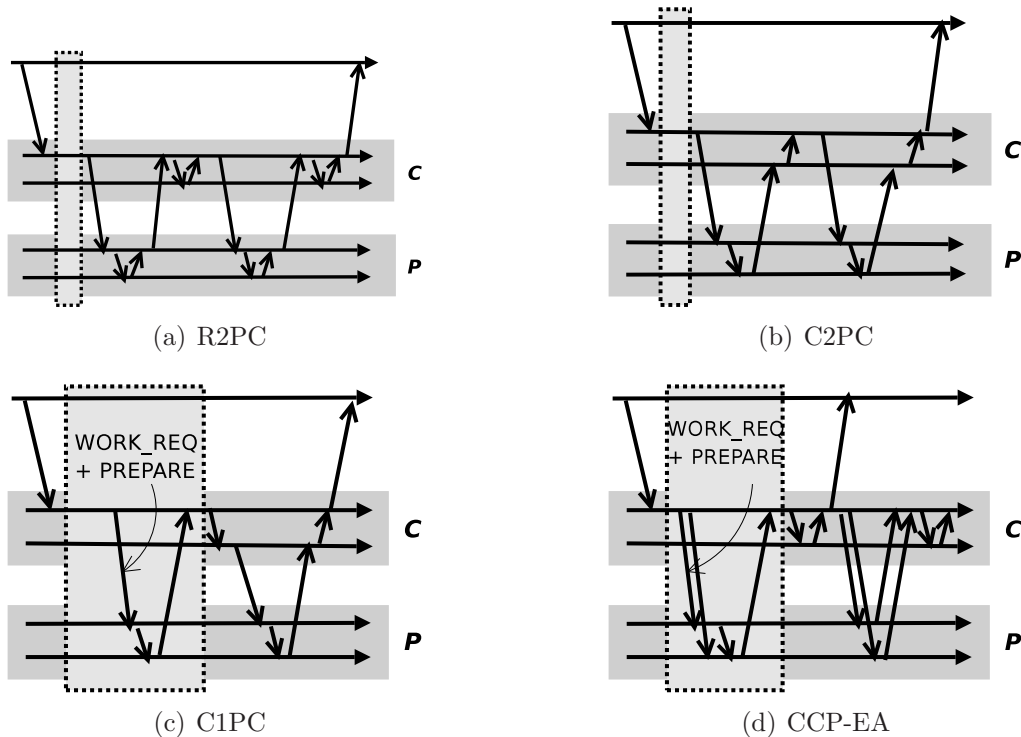


Figure III.2: Main-memory commit protocols

### III.3.3 Circular Two-Phase Commit

The *Circular Two-Phase Commit Protocol* (C2PC), as presented by the authors [19], is a protocol designed especially for main-memory primary-backup systems. As the name indicates, it has the same two phases as 2PC and R2PC. However, as shown in Fig. III.2(b) the commit processing is executed in a circular fashion to reduce the number of messages. The *primary coordinator* initiates the commit processing by sending **Prepare** to all *primary participants*. Each *primary participant* piggybacks its vote on a **Prepare** message to its corresponding *backup participant*. Each of the *backup participants* piggybacks its vote on a **Prepare** message to the *backup coordinator*,

which collects the votes and makes a decision.

In the decision phase, the decision is sent to the *primary coordinator* and subsequently propagated to the primary and backup participants in the same fashion as the **Prepare** message. The *backup participants* acknowledges the decision by sending **Commit** to the *backup coordinator*. After the acknowledgements have been sent, the participants can forget about the transaction. Finally, the *primary coordinator* receives **Committed** from the *backup coordinator* and the transaction is completed.

A more detailed explanation and proof of correctness for C2PC is given in [19].

### III.3.4 One-Phased Protocols

One-phased protocols skip the voting phase by making assumptions about the participants which can cause severe restrictions on the execution of transactions [1]. The oldest is the *Unsolicited Vote protocol* (UV) outlined by Stonebraker [40] where **Prepare** is piggybacked on **DoWork**. Thus, the voting phase is included in the transaction processing. Again, a simple version for main-memory primary-backup systems is developed. In *Replicated One-Phase Commit Protocol*, R1PC, the disk writes have been replaced by sending messages to the backups [19].

#### Circular One-Phase Commit

The *Circular One-Phase Commit Protocol* (C1PC) is a one-phased version of C2PC, based on UV, as presented by the authors [19]. Fig. III.2(c) illustrates the protocol. The *backup participants* send their votes to the *primary coordinator* instead of the *backup coordinator* as in C2PC. This is done since the *primary coordinator* may need to do additional work after the results have been received and before preparing the transaction locally. Thus, the *primary coordinator* collects the votes and makes a decision which is sent to the *backup coordinator*. It is responsible for distributing the decision to the *primary participants*. From there on, C1PC works as C2PC.

#### ClustRa Commit Protocol

The commit protocol used in ClustRa [16], uses a one-phased main-memory commit protocol. Fig. III.2(d) illustrates the execution. The *primary coordinator* sends out the work requests and piggybacks the prepare message to all *primary participants*. For each *backup participant* it sends out the number of expected log records to receive from the corresponding *primary participant*.

When the backup has received the log records, the vote is returned to the *primary coordinator*. The *primary coordinator* persistently stores the decision and identifiers of the participants to the *backup coordinator* and give an *early answer* to the client. Then, all *participants* receive a **Commit**, and reply with **Committed**. The successful completion of the commit processing is then persistently stored at the *backup coordinator*, before the transaction is completed and forgotten.

The *Early Answer* optimization, EA, used by ClustRa is applicable to the other protocols as well. Generally, a transaction coordinator does not need to wait until the end of the transaction before it replies to the client of the transaction. It only needs to wait until the decision to commit or abort is persistently stored. For 2PC, the reply can be given once the decision is saved to disk and for C2PC, R2PC, C1PC and R1PC the reply can be sent as soon as both the *primary* and *backup coordinators* know the outcome of the transaction. Hence, the time to execute the decision phase of the commit processing is not included in the response time. Thus, the response time of a transaction, as seen by clients, is reduced at no extra costs.

In this paper, we refer to the ClustRa commit protocol without the early-answer optimization as CCP to keep with the syntax of the other optimizations. The protocol as presented in [16] includes, however, the early-answer optimization and is labeled CCP-EA from here on.

CCP normally piggybacks some acknowledgement messages on other messages, therefore reducing overhead. The other protocols can employ the same technique. However, this paper does not consider the effects of piggybacking. Thus, CCP is also evaluated without piggybacking to ensure a fair comparison.

## III.4 Dynamic Coordinators

When executing distributed transactions, the number of participating nodes should be kept at a minimum. This can be achieved by dynamically assigning the coordinator tasks to one of the participants. This section presents some new ways to coordinate transaction processing in such a way that the overhead from sending a transaction to a node which is not a participant is limited.

### III.4.1 Dynamic Backup Coordinator

One way to reduce the overhead is to make one of the backup participants the backup coordinator of the transaction. This optimization is called the

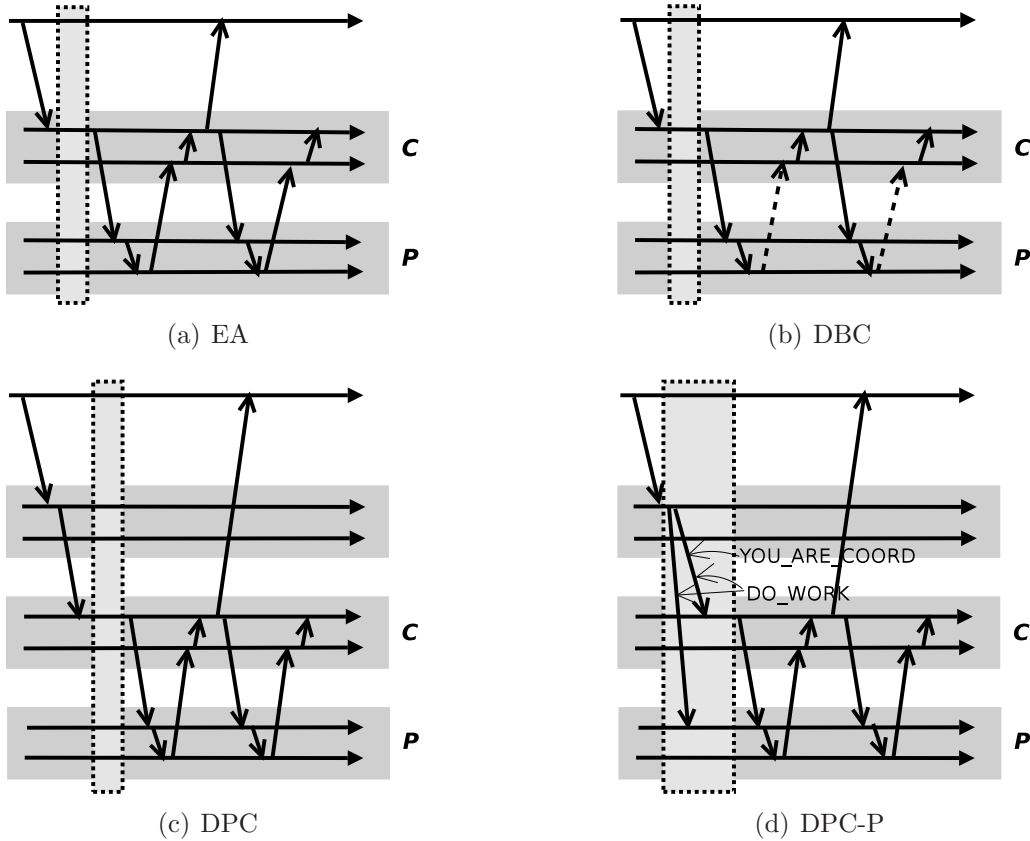


Figure III.3: Execution of C2PC with the dynamic coordinators optimization and giving early answers

*Dynamic Backup Coordinator*, DBC. This will reduce the overhead for C1PC and C2PC, but not for the general commit protocol. DBC is illustrated in Fig. III.3(b). The dotted arrows are the saved commit processing messages between the chosen *backup participant* and the *backup coordinator*. Also, the combined *backup coordinator* and *backup participant* prepares and commits as one entity and not as two separate ones. This saves two messages, one prepare task and one commit task. Clearly, since it is a fixed reduction of cost, this approach yields a better relative improvement for few participants, than for many. In Fig. III.3(b) and during the simulations, DBC are executed with EA.

### III.4.2 Dynamic Primary Coordinator

Another optimization is to forward the coordination of each transaction to a primary participant. Thus, one of the *primary participants* is guaranteed

to be the *primary coordinator* and the corresponding *backup participant* is the *backup coordinator*. This optimization is called the *Dynamic Primary Coordinator*, DPC. The forwarding of `BeginTxn` is shown in Fig. III.3(c). The performance gain is a complete round of commit processing for one node, minus the extra overhead of forwarding the transaction. As with DBC, the relative improvement is increased as the number of participants decreases.

DPC is applicable to all commit protocols described in Section III.3. In Fig. III.3(c) and during the simulations DPC use EA.

### III.4.3 Dynamic Primary Coordinator using Piggybacking

DPC can be further improved. Say, a node which is not a participant of the transaction receives a `BeginTxn`. Instead of just forwarding it to a participant it can send `DoWork` to all participants. To one of the participants it can set a `YouAreCoordinator` flag, and to the others it can set a `CoordinatorIs` variable to the address of the new coordinator. This approach is called the *Dynamic Primary Coordinator using Piggybacking*, DPC-P, and is shown in Fig. III.3(d). Thus, the extra overhead of forwarding the transaction is removed compared to the dynamic primary coordinator approach. This optimization will not add complexity to the failure semantics as all participants know the identities of the chosen primary and backup coordinator. As with DBC and DPC, the relative improvement is greater for few participants. In the figure and during the simulations DPC-P are executed with EA.

## III.5 The Simulation Model

To be able to evaluate the performance of various main-memory commit protocols, a realistic simulator of a distributed transaction processing system had to be developed. It has been implemented using Desmo-J, a framework for discrete-event modelling and simulation [33].

Measurements to provide the input values for the simulations are performed on an AMD Athlon(TM) 64 bits 3800+ processor, running a Linux 2.6 kernel. The length of the transaction operations are derived from measurements performed on a Java main-memory database prototype developed by Løland and Hvasshovd [28].

The distributed transaction processing (TP) system considered in this paper is composed of  $N$  nodes connected through a communication network, as shown in Fig. III.4(a). Each node has a *transaction manager* (TM) which is responsible for orchestrating the correct *local* execution of transactions. The

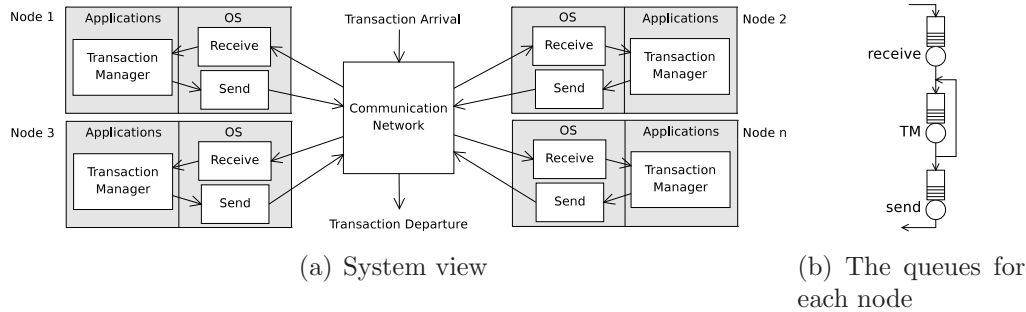


Figure III.4: The logical model of the system

correct *global* execution of transactions is ensured by coordination between TMs.

Each transaction is associated with a *coordinator*, which is responsible for execution and termination. Typically, the coordinator is the *transaction manager* (TM) at the node where the transaction was *initiated*. After the transaction execution is finished it is terminated using an atomic commitment protocol between the coordinator and participant TMs.

The distribution of processing and data is transparent to the transaction clients. The coordinator is responsible for issuing subtransactions to the appropriate nodes. Nodes receiving subtransactions are called *participants*. Generally, subtransactions can create further subtransactions, causing a multi-level transaction execution tree [29]. However, this paper discusses only trees that are one level deep. Transactions are assumed to be precompiled, and all nodes know where data are located. Thus, the coordinator can forward subtransactions directly to the correct participant.

We model each node to consist of a *transaction manager* (TM) executing on top of the operating system. The *operating system* (OS) is responsible for receiving and sending messages, while the transaction manager processes the requests. Other applications are assumed to be invoked from the TM. For each node, there are three First-In-First-Out queues: One for incoming messages, one for outgoing messages and one for tasks to be processed by the TM. This is shown in Fig. III.4(b). A processed task, can result in a new local task, which is inserted at the end of the TM-queue, or a remote task, which is inserted into a message and sent by the operating system.

Process *context switches* between the two processes of the system, OS and TM, are modelled. The OS alternates between sending and receiving messages. Measurements show that these are performed in just  $3.5 \mu\text{s}$ . The *timeslice* is the maximum time given to each process to execute requests before another process is given time at the cpu. The default timeslice for Linux kernel 2.6 systems is 100 ms and the minimum is 5 ms. To avoid

Table III.1: The input parameters of the simulation

Parameter	Value	
Simulation Model	Open	
Context Switch	3.5 $\mu$ s	
Timeslice	5 ms	
Send Message	40 $\mu$ s	
Receive Message	80 $\mu$ s	
Long Operations	700 $\mu$ s	DoWork, DoWorkAndPrepare
Medium Operations	150 $\mu$ s	BeginTxn, Prepare, Abort, Commit
Short Operations	50 $\mu$ s	WorkDone, Vote, Aborted, Committed
Simulation Time	200 s	
Capture Time	160 s [40 - 200]	
Simulation runs	10	
Subtransactions	3	
Transactions	Single tuple update	
# of simulated nodes	20	
CPU Utilization	Variable	

delaying operations too long for this application, a timeslice of 5 ms is chosen.

An *open* simulation model is used. The utilization of the system were varied from 1% - 97.5% to see the impact on the system performance. To be able to set the utilization for each protocol and optimization a single transaction was run for each combination, and the total service demand of it was found. Then, since there are only a few context switches per transaction at high utilization, the costs of nearly all context switches was subtracted from the total. Then, the altered service demand was used as a divisor to find the maximum throughput for each of the protocols per node.

Each transaction consists of three subtransactions, each of which containing a simple update operation. Thus, during transaction processing, each subtransaction makes one visit to its respective participant and returns. The subtransactions are processed in parallel. The coordinator and participants are chosen randomly and uniquely for each transaction. Since none of them are the same, each of the three subtransactions of a transaction has a unique destination node.

In line with update transactions, the messages sent over the network are assumed to be small, i.e. 200 - 300 bytes. Test experiments on a 100 Mbit/s Ethernet network show that no queuing effects for the network are likely to

happen for the load of a distributed main-memory database. Also, small messages have a very short transmit time. Thus, the time to send a message from one node to another can be modelled as processing time at the receiver and sender. Measurements of CPU usages indicate that it takes twice as long to receive as to send a message.

Operations are divided into three groups: Long, medium, and short. These are made to reflect the various service demands needed by an *average* operation of that kind. Long operations (`DoWork` and `DoWorkAndPrepare`) take 700  $\mu$ s, medium operations (`BeginTxn`, `Prepare`, `Abort` and `Commit`) take 150  $\mu$ s, and short operations (`WorkDone`, `Vote`, `Aborted` and `Committed`) take 50  $\mu$ s.

As the primary focus of this paper is the performance of commit protocols, the database internals are not modelled. A very large number of records is assumed, giving negligible data contention, hence, lock waiting effects have been ignored.

Each simulation lasted for 200 simulated seconds. The first 40 seconds are disregarded, as we are interested in the *steady-state* system. For each result presented, 10 simulation runs with random seed generators were made.

Table III.1 summarizes the simulation parameters.

## III.6 Simulation Results

Using the system model described in Section III.5, simulations were performed. Each of the optimizations, i.e. none, EA, DBC, DPC, and DPC-P, were combined with each of the protocols. The results of these simulations are presented here. First, graphs for the average response time versus throughput per node for each optimization are shown. Second, the common requirement of an upper bound on the maximum response time for 95% of the transactions [16] is explored for the best optimization. Third, the distribution of the response times is investigated.

### III.6.1 Average Response Time

Fig. III.5 plots the response time versus the throughput for each combination of protocol and optimization. No optimizations are plotted as diamonds, EA as triangles pointing down, DBC as triangles pointing up, DPC as squares, and DPC-P as circles. Fig. III.5(a) shows C2PC in black and R2PC in white, and Fig. III.5(b) shows C1PC in black, R1PC in white, and CCP in grey.

The response time of a transaction is the time from a client starts to send the request to the system, until it receives a reply. For standard transaction

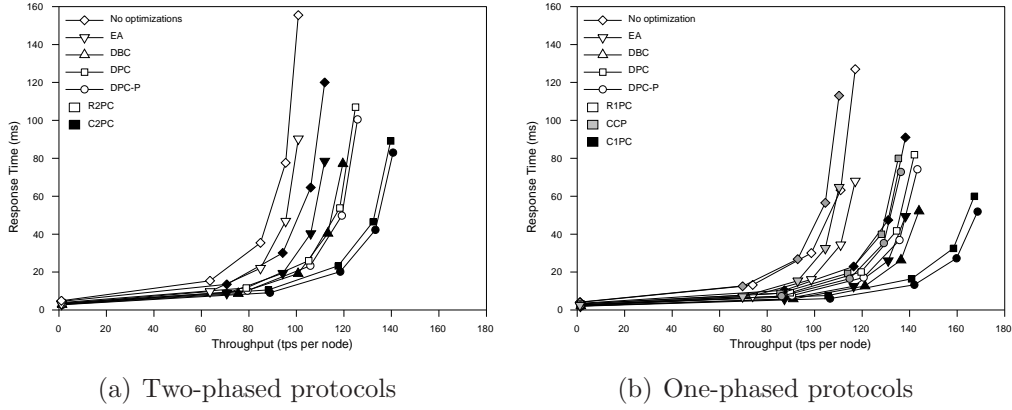


Figure III.5: Average response times simulating the dynamic coordinator optimizations

processing, the system does not reply to the client until the commit processing has completed. Comparing the protocols using no optimizations, C2PC gives a maximum throughput of around 11% more than R2PC and 2% more than CCP. C1PC results in a 38%, 26%, 25% and 19% increase in throughput compared to R2PC, CCP, C2PC and R1PC, respectively.

The Early Answer (EA) optimization replies earlier in the transaction processing. The maximum throughput is the same as not using any optimization since the total amount of work is the same. The average response times, however, are significantly reduced by 32-40% for C2PC and R2PC, 40-46% for C1PC and R1PC, and 38-44% for CCP.

The Dynamic Backup Coordinator approach (DBC) was simulated in conjunction with EA. Compared to just using EA, it reduces the response time and increases the maximum throughput with about 4% and 7% for C1PC and C2PC, respectively. This reduction is caused by avoiding some processing at and communication between the joint *backup participant* and *backup coordinator* node. However, this optimization does not apply to R2PC, R1PC and CCP since there is no direct communication between the *backup participants* and the *backup coordinator*.

The Dynamic Primary Coordinator approach (DPC) was used in conjunction with EA for these simulations. Compared to only using EA, the response times increase by 20 - 30% for utilizations up to 60%, but is more effective for higher utilizations. The reason for the increase at lower utilizations is the need for the processing at the node forwarding the transaction. The maximum throughput is increased by 20 - 25% by using DPC in addition to EA. CCP has a better response time than C2PC and R1PC for utilizations below 80%, but because of higher total overhead, the maximum throughput

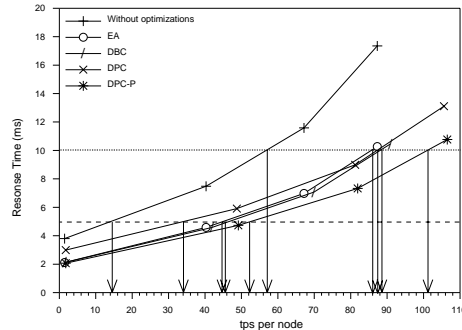


Figure III.6: The maximum response time for the 95% quickest transactions for the C1PC optimization. 20 nodes are simulated and the plots are cut off at 60% utilization

is lower. Again, R2PC is the worst performer and C1PC is the best.

The Dynamic Primary Coordinator and Piggybacking YouAreCoord approach (DPC-P) was simulated with EA. The results from the DPC-P simulation is marginally better than the results from DPC. The maximum throughput increases about 1% for each of the protocols and the response time is slightly reduced since there is less delay before sending out the work requests to the participants.

The differences in throughput between executing transactions with no optimization or EA and transactions using DPC-P are substantial. Looking at the differences in maximum throughput for the optimizations shows the improvement. R2PC and C2PC improves by 25%. C1PC improves by 22%, while R1PC and CCP improves by 23%.

Comparing the best in Fig. III.5, C1PC - DPC-P, to the worst, R2PC, and the existing MM protocol, CCP - EA, an increase of throughput of 68% and 45% is achieved.

### III.6.2 Response Time Demands

For real-time databases, a typical requirement is to have 95% of the transactions to respond within a given constraint [16]. Fig. III.6 shows the maximum response time for the 95% quickest transactions for all protocols using the best performing protocol, C1PC. This means that only 5% of the transaction responded slower for each of the plots. In the figure, the 10 ms and 5 ms response time is marked with a dotted and dashed line, respectively, and arrows mark the limit in throughput for each of the protocols.

The long arrows in Fig. III.6 shows that for C1PC - DPC-P tolerate approximately 80% more transactions than not using any optimizations before

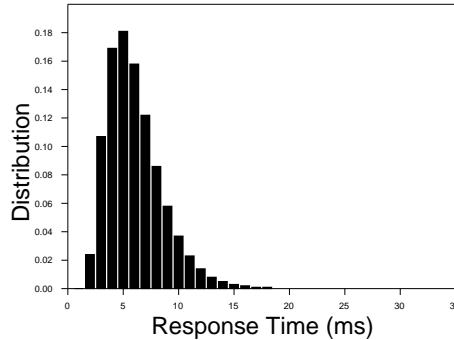


Figure III.7: Distribution of response times for C1PC with the DPC-P optimization at 60% utilization

crossing the 10-milliseconds limit for more than 5% of the transactions. Compared to EA, DBC, and DPC the improvement is around 15%. The short arrows illustrate the improvements for DPC-P versus no optimizations, DPC, EA, and DBC for a 5 ms response time limit, are about 250%, 50%, 25%, and 25%, respectively. This indicates that the relative performance of C1PC over the others are increasingly getting better as system response time requirements get more stringent. For short response time requirements, DPC is performing worse than DBC and EA because of the extra overhead of forwarding the transaction to the correct node.

### III.6.3 Distribution

The response time distribution for C1PC using DPC-P at 60% utilization, or 106 tps/node, is shown in Fig. III.7. Each response time have been rounded to the nearest integer value and the number of transactions for each are counted. Almost all reply before 15 ms, and approximately 94% reply within 10 ms. This fits well with the plot for C1PC in Fig. III.6, which shows that, for a load of 101 tps/node, 95% of the transactions reply within 10 ms.

## III.7 Analysis

To verify the results from the simulations, an analytical model of the system was constructed. The transaction workload is split into individual operations (or classes), such as `BeginTxn`, `DoWork`, `Commit`, `Send`, etc. Also, there is an unlimited number of clients of the system. This gives a multiple-class, open-queuing network.

A Poisson process generates requests with exponentially distributed inter-arrival times. The number of operations per transaction are counted for each

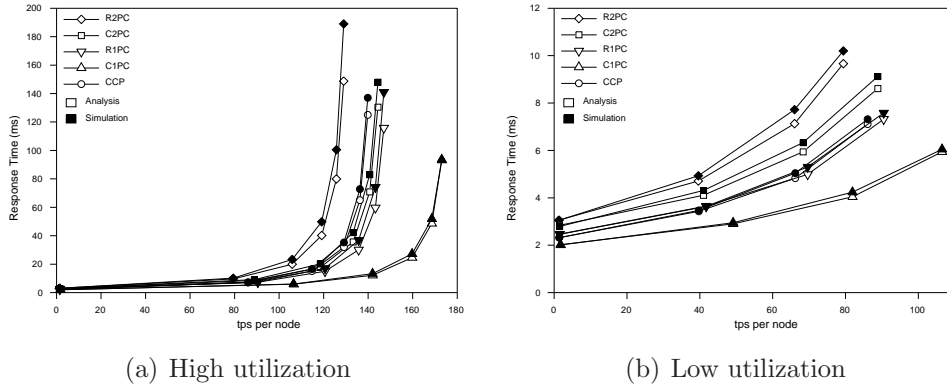


Figure III.8: Comparing analysis and simulations for all protocols using DPC-P

type of protocol to give the relative frequency of each of the classes. The load added by context switches are treated as a separate workload class. Its frequency is found by using the output from the simulations and counting the number of context switches performed for each combination of protocol and utilization.

Assuming uniformity across the nodes, single-server queueing theory [17] can be used. This is a single resource, with a single queue in front. The Poisson process generates at an arrival rate given by the sum of the arrival rates from each class for the given protocol. The steady-state solutions are derived using results from  $M/G/1$  queues [17]. The resulting total residence time is then weighed against the number of serial requests of each class. To find the slowest response time for the three subtransactions, numerical Laplace inversion of waiting times of  $M/D_N/1$  queues [37] is used.

Fig. III.8 shows a comparison of the analyses and simulations of 20 nodes using the best performing enhancer, DPC-P. R2PC is plotted as diamonds, C2PC as squares, R1PC as triangles pointing down, C1PC as triangles pointing up, and CCP as circles. The analysis marked as white, while the simulations are shown in black. There are some discrepancies between the analysis and simulation. These are caused by some factors from the simulations that are not covered by this analysis: (1) The priority effect as introduced by the scheduling policy and (2) the number of nodes. The first slightly increases the simulation response time since some of the operations are delayed waiting for the 5-millisecond timeslice to run out before the context switch. For the second, when varying the number of nodes, the differences in response time are quite large. The response time analysis for all protocols are between the simulation of 10 and 50 nodes. These issues have been further explored in [20]. Despite the differences shown in this analysis, the shape of the anal-

ysis plots seems to verify the general shape of the simulation plots for high utilization, as shown in Fig. III.8(a).

Fig. III.8(b) compares the simulation and analysis for utilizations below 60%. This is in the range where a system normally operates and the maximum difference here is less than 7%. Thus, the simulations are clearly well suited to the analysis for normal operating conditions.

## III.8 Evaluation and Conclusion

This paper has presented three new optimization techniques in the context of main-memory commit protocols. The optimizations have been paired with existing main-memory protocols. Also, the protocols and optimizations have been simulated and the results have been verified analytically. The results show that when applicable, one-phased protocols perform better than two-phase protocols, with C1PC as the best one. C2PC was found to be the best two-phased protocol.

The dynamic coordinator optimizations improves the response time and throughput even further. The best performing combination of protocol and optimization was found to be C1PC using the DPC-P optimization.

For further work, a full scale implementation of the best protocol and optimizations should be incorporated in an existing main-memory system. It would show the effect of these protocols and optimizations for real-world applications. Also, an analytical model where the effects of varying the number of nodes are shown should be developed. Further, the effects of piggybacking messages on another with the same destination can be investigated.

## References

- [1] Maha Abdallah, Rachid Guerraoui, and Philippe 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] Maha Abdallah and Philippe Pucheral. A single-phase non-blocking atomic commitment protocol. In *DEXA '98: Proceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 584–595, London, UK, 1998. Springer-Verlag.
- [3] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency con-

- trol performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.
- [4] Y. J. Al-Houmaily and P. K. Chrysanthis. Two-phase commit in gigabit-networked distributed databases. In *Proceedings of the 8<sup>th</sup> ISCA International Conference on Parallel and Distributed Computing Systems*, pages 554–560, 1995.
- [5] Yousef Jassem Al-Houmaily. *Commit processing in distributed database systems and in heterogeneous multidatabase systems*. PhD thesis, University of Pittsburgh, Pittsburgh, PA, USA, 1997.
- [6] B. S. Boutros and B. C. Desai. A two-phase commit protocol and its performance. In *DEXA '96: Proceedings of the 7th International Workshop on Database and Expert Systems Applications*, page 100, Washington, DC, USA, 1996. IEEE Computer Society.
- [7] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [8] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, New York, NY, USA, 1984. ACM Press.
- [9] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 04(6):509–516, 1992.
- [10] Dieter Gawlick and David Kinkade. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985.
- [11] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.
- [12] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154. IEEE Computer Society, 1981.
- [13] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

- 
- [14] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. Revisiting commit processing in distributed database systems. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 486–497, New York, NY, USA, 1997. ACM Press.
- [15] Jayant R. Haritsa, Krithi Ramamritham, and Ramesh Gupta. The PROMPT real-time commit protocol. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):160–181, 2000.
- [16] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, 1995.
- [17] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley & sons, 1991.
- [18] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Sergio Arévalo. A low-latency non-blocking commit service. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 93–107. Springer-Verlag, 2001.
- [19] Heine Kolltveit and Svein-Olaf Hvasshovd. The Circular Two-Phase Commit Protocol. In *Proceedings of International Conference of Database Systems for Advanced Applications*, pages 249–261. Springer-Verlag, 2007.
- [20] Heine Kolltveit and Svein-Olaf Hvasshovd. Performance of Main Memory Commit Protocols. Technical Report 06/2007, NTNU, IDI, 2007.
- [21] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th Conference on Very Large Databases*. Morgan Kaufman, 1993.
- [22] Butler W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265, London, UK, 1981. Springer-Verlag.
- [23] Inseon Lee and Heon Young Yeom. A fast commit protocol for distributed main memory database systems. In *ICOIN '02: Revised Papers from the International Conference on Information Networking, Wireless Communications Technologies and Network Applications-Part II*, pages 691–702, London, UK, 2002. Springer-Verlag.

- 
- [24] Inseon Lee and Heon Young Yeom. A single phase distributed commit protocol for main memory database systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 44–51, Washington, DC, USA, 2002. IEEE Computer Society.
- [25] Inseon Lee, Heon Young Yeom, and Taesoon Park. A New Approach for Distributed Main Memory Database Systems. *IEICE Transactions on Information and Systems*, E87D(1):196–204, January 2004.
- [26] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. An optimistic commit protocol for distributed transaction management. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, pages 88–97, New York, NY, USA, 1991. ACM Press.
- [27] M. L. Liu, D. Agrawal, and A. El Abbadi. The performance of two phase commit protocols in the presence of site failures. *Distributed and Parallel Databases*, 6(2):157–182, 1998.
- [28] Jørgen Løland and Svein-Olaf Hvasshovd. Online, non-blocking relational schema changes. In *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 405–422. Springer Berlin, 2006.
- [29] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R\* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.
- [30] C. Mohan and Bruce G. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 76–88, 1983.
- [31] Matthias Nicola and Matthias Jarke. Performance modeling of distributed and replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):645–672, 2000.
- [32] P. K. Chrysanthis and G. Samaras and Y. J. Al-Houmaily. *Recovery and Performance of Atomic Commit Processing in Distributed Database Systems*, chapter 13, pages 370–417. Prentice Hall, 1998.
- [33] Bernd Page and Wolfgang Kreutzer. *The Java Simulation Handbook. Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 2005.

- 
- [34] Taesoon Park and Heon Y. Yeom. A consistent group commit protocol for distributed database systems. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS'99)*, 1999.
- [35] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 520–529, Washington, DC, USA, 1993. IEEE Computer Society.
- [36] George Samaras, George K. Kyrour, and Panos K. Chrysanthis. Two-phase commit processing with restructured commit tree. In *Proceedings of the Panhellenic Conference on Informatics*, volume 2563, pages 82–99. Springer-Verlag, 2003.
- [37] John F. Shortle, Martin J. Fisher, and Percy H. Brill. Waiting Time Distribution of  $M/D_N/1$  Queues Through Numerical Laplace Inversion. *INFORMS Journal on Computing*, to appear.
- [38] James W. Stamos and Flaviu Cristian. A low-cost atomic commit protocol. In *Proceedings of the Ninth Symposium of Reliable Distributed Systems*, 1990.
- [39] James W. Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4), 1993.
- [40] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.
- [41] Alexander Thomasian. Concurrency control: methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.
- [42] Y.J. Al-Houmaily and R. Conticello and J.Pike and P.K. Chrysanthis. Performance of Five Atomic Commit Protocols in Gigabit-Networked Database Systems. In *Proceedings of International Conference on Parallel and Distributed Computing Systems*, pages 29–36, 1998.

# Paper IV

Main-Memory Commit Processing: The Impact of Priorities.

Heine Kolltveit and Svein-Olaf Hvasshovd.

In *Advances in Databases; Concepts, Systems and Applications*, SpringerLink.

March 19 - 21, 2008, New Delhi, India.



# Main-Memory Commit Processing: The Impact of Priorities

Heine Kolltveit and Svein-Olaf Hvasshovd

Department of Computer and Information Science  
Norwegian University of Science and Technology  
NO-7491 Trondheim, Norway

## Abstract

Distributed transaction systems require an atomic commitment protocol to preserve ACID properties. The overhead of commit processing is a significant part of the load on a distributed database. Here, we propose approaches where the overhead is reduced by prioritizing urgent messages and operations. This is done in the context of main-memory primary-backup systems, and the proposed approaches are found to significantly reduce the response time as seen by the client. Also, by piggybacking messages on each other over the network, the throughput is increased. Simulation results show that performance can be significantly improved using this approach, especially for utilizations above 50%.

## IV.1 Introduction

The increasing complexity of modern computer systems puts increasing demands on the performance of all components of systems. Databases are commonly an important part of these systems. Thus, it is crucial that they are fast and reliable. The ratio between storage capacity and cost of main memory (MM) have been increasing exponentially, which has contributed to the advent of MM databases and applications. Since the data in MM databases is always in MM, slow disk accesses are replaced with faster MM accesses to data [6, 7], improving the potential performance.

MM databases can be classified by how *logging* and *persistence* are handled. The first determines whether the log is saved to disk or not. The latter dictates whether the log is persistently stored or not. If the log is

saved to disk, persistence is achieved if it is on disk before the transaction commits. If it is only stored in MM, it must be replicated to a process with a different failure mode [5] to make it persistent. Here we discuss only *pure* main-memory systems, where neither data nor log resides on disk. Both data and the log are replicated to achieve persistence.

*Atomic commitment protocols* are designed to ensure that all participants of a transaction reach the same decision about a transaction. As soon as the decision is reached and has been persistently stored, an *Early Answer* [11] can be given to the client. This has been shown to significantly reduce transaction response time [15].

This paper is motivated by the observation that operations (or tasks) and messages before the early answer are more urgent than those after. Also, larger messages take less time to send and receive per byte than smaller messages. Thus, by *piggybacking* messages from a single source to a common destination, the transmission of messages is more effective.

The main contributions of this paper are the usage of priorities and piggybacking in a main-memory distributed database setting and simulation results showing the performance benefits gained by using this approach. The results of such an evaluation can favorably be used by system developers to improve the performance of MM applications, e.g., in a shared-nothing fault-tolerant DBMS like ClustRa [11]. In addition, the simulation framework developed in [15] has been improved to facilitate piggybacking and priorities.

This paper only simulates failure-free execution where all transactions commit. Since this is the dominant execution path, it is where the potential for performance gains is greatest.

The rest of the paper is organized as follows. Section IV.2 gives an overview of related work. Section IV.3 presents the priority and piggybacking schemes. Section IV.4 outlines the simulation model and parameters used for the simulations in Section IV.5. The simulations are compared to a statistical analysis in Section IV.6, while Section IV.7 gives some concluding remarks.

## IV.2 Related Work

Several 2PC-based [9, 18] modifications exist where performance issues are handled [21]. Presumed commit and presumed abort [18] both avoid one flushed disk write, by assuming that a non-existent log record means that the transaction has committed or aborted, respectively. Transfer-of-commit, lazy commit and read-only commit [10], sharing the log, [18, 22] and group commit [8, 20] are other optimizations. An optimization of the presumed

commit protocol [16] reduces the number of messages, but requires the same number of forced disk writes.

One-phased commit protocols have also been proposed [22, 2, 1, 17, 23, 11]. These are based on the early-prepare or unsolicited-vote method by Stonebraker [24] where the prepare message is piggybacked on the last operation sent to a participant. In this way, the voting phase is eliminated. However, these approaches may inflict strong assumptions and restrictions on the transactional system [1].

1-2PC [3, 4] is an approach which dynamically chooses between one-phased execution and two-phased. Thus, it provides the performance advantages of 1PC when applicable, while also supporting the wide-spread 2PC.

There exists some pure main memory commit protocols that ensure persistence. R2PC and R1PC use synchronous messages to backups to persistently store the log, while C2PC and C1PC executes the commit processing in a circular fashion [13, 15]. The ClustRa commit protocol [11], CCP, is another protocol. It is a highly parallel one-phased protocol. R2PC, R1PC, C2PC, C1PC and CCP are the protocols used in this paper. See [14] for a more thorough presentation of them. A dynamic coordinator algorithm called DPC-P [15] together with the Early Answer optimization [11] is also used in this paper.

## IV.3 Priorities and Piggybacking

The early-answer optimization allows us to divide messages and tasks into two separate groups: The ones before the reply and the ones after. Intuitively, to achieve a short response time, the ones before the reply should be prioritized. This section first presents a scheme for prioritizing operations and then three schemes for prioritizing and piggybacking messages. Pseudocode for the algorithms is fully presented in [14].

Prioritizing messages before the reply leads to longer locking of data at the participants. However, we assume a large number of records and minimal data contention. Therefore, the extra time spent before releasing the locks does not degrade the system.

### IV.3.1 Priority of Tasks by the Transaction Manager

All nodes are assumed to have a transaction manager, which manages the transactional requests. In front of it there is a queue of tasks. An **urgent** flag is set for the task, if it is an operation that happens before the reply to the client. When the transaction manager is ready to perform a task, it

starts at the front of the queue and checks if the first task is urgent. If it is, it is immediately processed. If not, the task's `waited` field is checked. If it has waited longer than a predetermined maximum waiting period, it is processed, otherwise the algorithm searches for the first urgent task in the queue. If none is found, the first in the queue is chosen. We call this priority by the transaction manager *PRI*. The default way without using *PRI* is called *NOPRI*.

### IV.3.2 Priority and Piggybacking Messages over the Network

The messages sent over the network can be prioritized similarly to the tasks at the transaction manager. In addition, messages can be grouped together using piggybacking. *Piggybacking* is a technique where messages going to the same destination are appended on each other. This reduces the total load on the system since there is quite a large initiation cost for messages, and the total cost per bit sent decreases as the number of bits increases.

The rest of this section presents four alternate ways to chose which messages to piggyback.

#### **PB1: No Piggybacking and no Priority**

In this first setup, the queue of outgoing messages are served on a first-come first-served basis. No messages are given priority and no piggybacking occurs. This is the default way to process outgoing messages.

#### **PB2: Piggybacking, but no Priority**

For PB2, the queue is served on a first-come first-served basis. However, all the messages in the queue which are going to the same destination are piggybacked on the message. Thus, the number of required messages is reduced. Intuitively, this increases the throughput and decreases the response time.

#### **PB3: Piggyback from Non-Urgent Queue**

In PB3, the non-urgent messages are moved to a separate queue. When an urgent message is found, the non-urgent queue is checked for messages with the same destination. These are piggybacked on the urgent message. To avoid starvation of non-urgent messages, a timeout is used. After the timeout expires for the first message in the non-urgent queue, it is sent, and the rest of the non-urgent queue is checked for messages to piggyback.

#### **PB4: Piggyback from Urgent and Non-Urgent Queue**

PB4 is similar to PB3, but extends it by piggybacking messages from both outgoing queues. Thus, more messages are potentially piggybacked on the same message. As for PB3, a timeout is used to avoid starvation of non-urgent messages.

## **IV.4 The Simulation Model**

To evaluate the performance of the commit protocols a discrete-event simulator was developed using Desmo-J [19]. The input parameters of the system are given in Table IV.1. The model and the parameters have been more extensively explained in [14]. However, the cost of sending and receiving messages has changed. In line with update transactions, the messages sent over the network are assumed to be small, i.e. 500 bytes for the first message to reach each participant for each transaction, and 50 bytes for the others. The formulas for the response time are found by taking the average of 100.000 round-trip times for UDP datagrams with 10 bytes intervals. Measurements of CPU usages indicate that it takes twice as long to receive, as to send a message.

## **IV.5 The Simulation Results**

The simulations were performed using the system model presented in Section IV.4. The results from executing the main-memory protocols and the chosen optimizations are presented in this section.

### **IV.5.1 Average Response Time**

Simulations were performed for all combinations of protocols, priority and piggybacking algorithms. The resulting average response times versus throughputs are plotted in Figure IV.1. The DPC-P optimizations have been chosen since it gives the best performance [15]. PB1, PB2, PB3, and PB4 is plotted as circles, squares, triangles pointing up, and triangles pointing down, respectively. NOPRI is shown in white, and PRI in black. The results for each protocol have been plotted for throughputs up to 97.5% utilization of the original protocol combined with the optimization not using any piggybacking (PB1) and NOPRI.

Figure IV.1 clearly show that PB1 with NOPRI is the worst performer. The PRI approach results in faster responses to the client, at the costs of

Table IV.1: The input parameters of the simulation

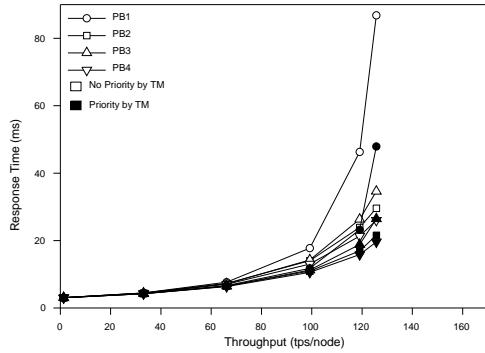
Parameter	Value
Simulation Model	Open
Context Switch	3.5 $\mu s$
Timeslice	5 ms
Send Message	$(26+0.055*B) \mu s$ if $B < 1500$ , $(73+0.030*B) \mu s$ else
Receive Message	$(52+0.110*B) \mu s$ if $B < 1500$ , $(146+0.060*B) \mu s$ else
Long Operations	700 $\mu s$   DoWork,DoWorkAndPrepare
Medium Operations	150 $\mu s$   BeginTxn,Prepare,Abort,Commit
Short Operations	50 $\mu s$   WorkDone,Vote,Aborted,Committed
Simulation Time	200 s
Capture Time	160 s [40 - 200]
Simulation runs	10
Subtransactions	3
Transactions	Single tuple update
# of simulated nodes	20
CPU Utilization	Variable

longer holding of the locks at the participants, since the commit messages might be delayed. At low utilizations, PB3 performs better than PB2 since the first prioritizes messages to send, while the latter only uses piggybacking. However, as the utilization increases and the queues grow, PB2 outperforms PB3, because PB3 will only send one prioritized messages each time, while PB2 might send more. PB4 does not have this problem since both the urgent and the non-urgent queues are checked for messages to piggyback.

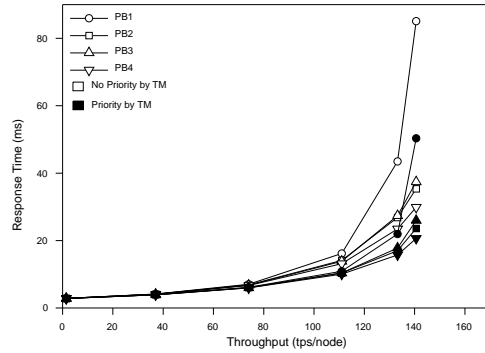
The combination of PRI and PB4 yields the best performance for all protocols, optimizations and utilizations. This is caused by the fact that it always piggybacks all messages in the outgoing queues going in the same direction and always prioritizes the most urgent tasks to be performed by the TM. Priority of tasks by TM has a greater impact on the performance than piggybacking for utilizations up to approximately 90%. From there on, piggybacking using PB4 has more effect.

## IV.5.2 Response Time Demands

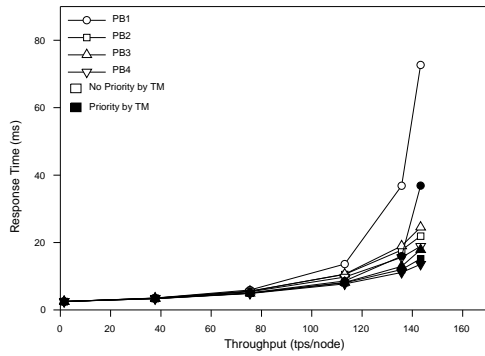
Real-time databases typically require that most (i.e. 90%, 95%, 97%, 98% or 99%) transactions reply within a given time limit [11]. Figure IV.2 plots the maximum of the 95% shortest response times using the best performing



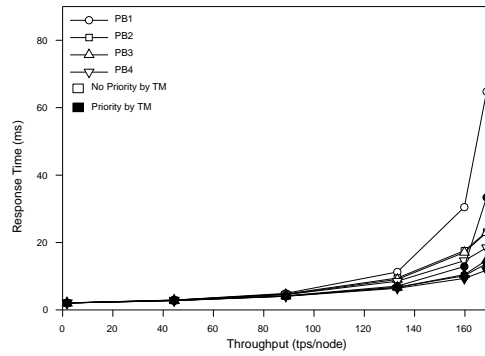
(a) R2PC



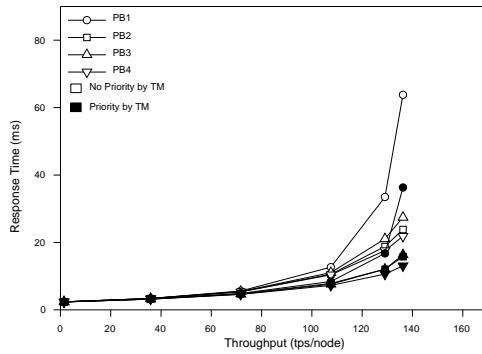
(b) C2PC



(c) R1PC



(d) C1PC



(e) CCP

Figure IV.1: The effect of varying the priority and piggybacking scheme for DPC-P-protocols

protocol, C1PC with the DPC-P optimization. The 5- and 10-millisecond limits are shown in dashed and dotted lines, respectively. The downward pointing arrows mark the throughput at the intersections between the simulations and the dashed and dotted lines.

The simulation results for C1PC - DPC-P are presented in Figure IV.2.

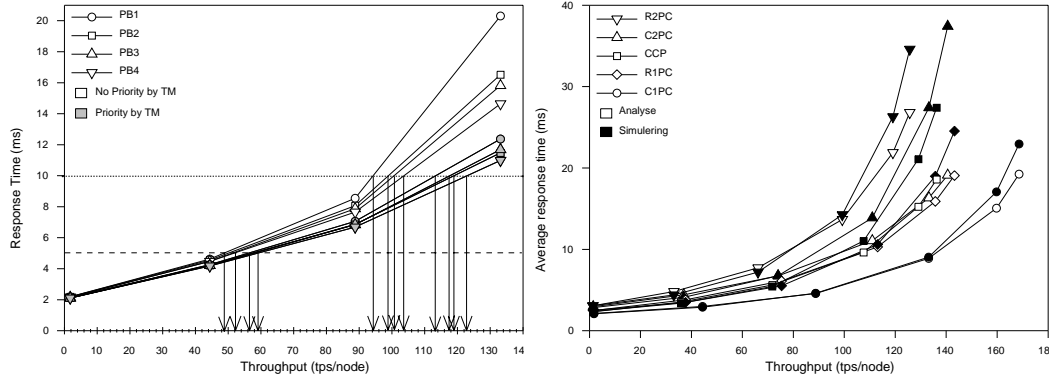


Figure IV.2: The maximum response time for the 95% quickest transactions while executing C1PC - DPC-P, DPC-P, NOPRI and PB3

At the 5-millisecond limit, PB4 tolerates 6% more transactions than PB1, both executed with NOPRI. For PRI, the same difference is 5%. The difference between executing with PRI and PB4, and with NOPRI and PB1 is 20%. For the 10-millisecond limit and NOPRI, PB2, PB3 and PB4 tolerate approximately 5% – 10% more transactions than PB1. With PRI, it is 6% – 9%. Comparing PB1 with and without PRI, a 20% improvement is found. The best combination, PB4 with PRI, tolerates 30% more transactions than the NOPRI and PB1 .

As for the average response time in Section IV.5.1, priorities give greater impact on performance of the protocols than piggybacking. Also, the simulations using PB4 and PRI shows the best performance.

## IV.6 Analysis

The simulation results are verified by using the analytical model used in [15]. The model uses a multiple-class open-queueing network with exponentially distributed inter-arrival times. It assumes uniformity for all nodes, such that single-server queueing theory [12] can be applied. The steady-state solutions are derived using results from  $M/G/1$  queues [12]. Refer to [14] for more details and results from the analysis.

There are some problems trying to analyse the piggybacking and priority approaches presented in Section IV.3. First, the dynamic sizes of the messages sent over the network is hard to estimate. The size is dependent on the number of messages waiting to be piggybacked on a message to a given destination. This number varies for distinct piggybacking algorithms. Also,

the non-urgent messages and tasks are being delayed both at the transaction manager and at the outgoing queue by the priority mechanism. However, PB3 and NOPRI can be approximated by assuming that all non-urgent messages are delayed until all other queues are empty. This works until the throughput reaches a limit where the non-urgent messages are sent because of the timeout. At this point, the performance of the simulation is degraded compared to the analysis.

Figure IV.3 illustrates the comparison of the analyses and simulations of 20 nodes using the DPC-P optimization, NOPRI, and PB3. The results from the analyses are shown in white and the simulations in black. R2PC, C2PC, CCP, R1PC, and C1PC are shown as triangles pointing down, triangles pointing up, squares, diamonds, and circles, respectively. Below about 80 to 110 tps/node, the analyses seem to verify the simulations. However, from there on, the effects of timeouts of the non-urgent messages as mentioned earlier are clear. The simulation times are increased, while the analyzes which do not include these effects show a lower response time.

## IV.7 Evaluation and Conclusion

This paper has presented new approaches to prioritize tasks and piggyback messages in the context of main-memory commit processing. The approaches have been evaluated by simulation using existing main-memory commit protocols and optimizations. The results show that the new approaches can improve the performance significantly for these protocols, especially for utilizations above 50%.

The best piggybacking algorithm approximately halves the average response time at 90% utilization compared to not using any piggybacking. The improvement is less for lower utilizations. Prioritizing messages has even more impact. At 90% utilization, the improvement in the average response time is around 56%. Combining those two, the total improvement in the average response time is more than two-thirds.

Using the response time requirement that 95% of the transactions respond within a time limit of 10 ms, a 23% and 30% improvement was found using piggybacking and priorities with DPC-P compared to not using these techniques for C2PC and C1PC.

For further work, the protocols, optimizations and piggybacking and priority approach should be implemented in a existing main-memory system. This would give results for real-world applications. Also, the effect of longer lock holding and the resulting increase in data contention caused by the priority approach should be investigated.

## References

- [1] Maha Abdallah, Rachid Guerraoui, and Philippe 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] Maha Abdallah and Philippe Pucheral. A single-phase non-blocking atomic commitment protocol. In *DEXA '98: Proceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 584–595, London, UK, 1998. Springer-Verlag.
- [3] Yousef J. Al-Houmaily and Panos K. Chrysanthis. 1-2PC: The one-two phase atomic commit protocol. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 684–691, New York, NY, USA, 2004. ACM Press.
- [4] Yousef J. Al-Houmaily and Panos K. Chrysanthis. ML-1-2PC: an adaptive multi-level atomic commit protocol. In *8th East-European Conference on Advances in Databases and Information Systems, ADBIS*, pages 275–290, September 2004.
- [5] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [6] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, New York, NY, USA, 1984. ACM Press.
- [7] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 04(6):509–516, 1992.
- [8] Dieter Gawlick and David Kinkade. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985.
- [9] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.

- 
- [10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [11] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, 1995.
- [12] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley & sons, 1991.
- [13] Heine Kolltveit and Svein-Olaf Hvasshovd. The Circular Two-Phase Commit Protocol. In *Proceedings of International Conference of Database Systems for Advanced Applications*, pages 249–261. Springer-Verlag, 2007.
- [14] Heine Kolltveit and Svein-Olaf Hvasshovd. Main memory commit processing: The impact of priorities - extended version. Technical Report 11/2007, NTNU, IDI, 2007.
- [15] Heine Kolltveit and Svein-Olaf Hvasshovd. Performance of Main Memory Commit Protocols. Technical Report 06/2007, NTNU, IDI, 2007.
- [16] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th Conference on Very Large Databases*. Morgan Kaufman, 1993.
- [17] Inseon Lee and Heon Young Yeom. A single phase distributed commit protocol for main memory database systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 44–51, Washington, DC, USA, 2002. IEEE Computer Society.
- [18] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R\* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.
- [19] Bernd Page and Wolfgang Kreutzer. *The Java Simulation Handbook. Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 2005.
- [20] Taesoon Park and Heon Y. Yeom. A consistent group commit protocol for distributed database systems. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS'99)*, 1999.

- 
- [21] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 520–529, Washington, DC, USA, 1993. IEEE Computer Society.
  - [22] James W. Stamos and Flaviu Cristian. A low-cost atomic commit protocol. In *Proceedings of the Ninth Symposium of Reliable Distributed Systems*, 1990.
  - [23] James W. Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4), 1993.
  - [24] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.

# Paper V

Main-Memory Commit Protocols for Multiple Backups.

Heine Kolltveit and Svein-Olaf Hvasshovd.

In *Proceedings of The Seventh International Conference on Networking*, IEEE.

April 13 - 18, 2008, Cancun, Mexico.



# Main-Memory Commit Protocols for Multiple Backups

Heine Kolltveit and Svein-Olaf Hvasshovd

Department of Computer and Information Science  
Norwegian University of Science and Technology  
NO-7491 Trondheim, Norway

## Abstract

Atomic commitment protocols are needed to preserve ACID properties of distributed transactional systems. The performance of such protocols is essential to avoid adding too much overhead to transaction processing. Also, many applications require levels of availability, which cannot be met by using only one backup source (disk or process). Combining these two requirements, this paper presents protocols for use with main-memory primary-backup systems, where there are multiple backup processes for each primary. These protocols are evaluated by simulation and verified by statistical analysis. The results show that the best choice of protocol is not static, but varies with the transactional load of the system.

## V.1 Introduction

Due to the increasing size of main-memory (MM) storage capacity and the corresponding decrease in cost, the applicability of MM databases widens. Since the database is a central component of computer systems, it is important that it is fast and reliable. MM databases log to a backup process instead of slower forced logging to disk, thus making it faster [6, 7]. This also ensures the reliability of both processing and data, since there are backups ready to take over the processing if one of the primary replicas fails.

MM databases can be classified according to their *log* and *persistence* policy. The first regulates whether the log is saved to stable storage or not. The latter determines whether the log is persistently stored or not. If the log policy states that the log should be saved to disk, the log is persistently saved only if a transaction's log records are flushed to disk before the transaction commits. On the other hand, if the log is only stored in MM, the log must be

replicated to another node with a independent *failure mode* [5]. This paper is in the context of *pure* MM databases, where both data and log is stored in MM only, and it is replicated to provide persistency.

An *atomic commitment protocol* ensures that all participants of a transaction reach the same decision. In pure MM systems, if a primary and its corresponding backup process fail at the same time, the system becomes inconsistent. However, for many applications this is acceptable and one backup process is sufficient. Other and more critical applications need a higher degree of fault tolerance. Thus, two or more backup processes are needed. As for systems with one backup [14] atomic commit protocols should be tailored for multiple backup systems.

In this paper, we present generalized versions of the protocols presented in [14, 15, 16]. Here, multiple backup processes can be attached to each primary. Also, simulations and analysis have been performed to evaluate the performance of the protocols. These results can favorably be used by system developers to improve the performance of distributed MM database systems such as ClustRa [11].

The rest of this paper is organized as follows. Section V.2 presents related work. The details of some existing MM protocols are shown in Sections V.3. Section V.4 generalizes these approaches to multiple backups. The simulation model in Section V.5 is used for the simulations performed in Section V.6. The results are compared to a statistical analysis in Section V.7. Finally, Section V.8 concludes the paper.

## V.2 Related Work

There exist many modifications of the two-phase commit protocol (2PC) [9, 19] which handles performance issues [23]. Presumed commit and presumed abort [19] both avoid one flushed disk write, by assuming that a non-existent log record means that the transaction has committed or aborted, respectively. Other optimizations include transfer-of-commit, lazy commit, and read-only commit [10], sharing the log [19, 24] and group commit [8, 21]. An approach [17] reduces the number of messages required by the presumed commit protocol.

Some one-phased protocols based on the early-prepare or unsolicited-vote protocol by Stonebraker [26] have also been proposed [24, 2, 1, 18, 25, 11]. In most of them, the prepare message is piggybacked on the work request, and the vote is piggybacked on the result of the work. Thus, the voting phase is included in the transactional processing. These approaches may, however, give strong restrictions on the transactional system [1].

1-2PC [3, 4] is an approach which dynamically chooses between one-phased execution and two-phased. Thus, it provides the performance advantages of 1PC when applicable, while also supporting the wide-spread 2PC.

Multiple backup coordinators have been proposed earlier to reduce blocking behaviour in 2PC [22], but it still uses disks as stable storage. In [13], a replicated group of commit servers is used to persistently save the decision. However, the participants are not replicated.

Commit protocols for replicated main-memory systems have been introduced by Kolltveit and Hvasshovd [14, 15]. These are presented in greater detail in Section V.3.

## V.3 Main-Memory Commit Protocols

This section briefly presents commit protocols for main memory primary-backup systems. The legend for Figure V.1 are given in Figure V.2(a).

### V.3.1 R2PC

The *Replicated Two-Phase Commit protocol* (R2PC) is an application of 2PC [9, 19] where the log records are sent to backups instead of flushing them to disk [14]. Figure V.1(a) illustrates the failure-free execution of R2PC. After the transactional operators have completed, the protocol is initiated by the *primary coordinator* which sends **Prepare** to all *primary participants*. Each of them stores its vote persistently at the corresponding *backup participant* before replying. The *primary coordinator* makes a decision based on the votes. A missing or negative vote results in a decision to **ABORT** the transaction, otherwise the decision is **COMMIT**. The decision is sent to the *backup coordinator*, which acknowledges it back to the *primary*. Then, the *primary* sends the decision to the *primary participants*. They either commit or abort the transaction based on the decision. Also, the decision is persistently stored by sending it to the *backup participants*. The *primary participants* send **Committed** or **Aborted** to the *primary coordinator* which sends it to the *backup coordinator* before replying to the client.

### V.3.2 C2PC

The *Circular Two-Phase Commit protocol* (C2PC) is a protocol tailored specifically for main-memory primary-backup systems [14]. While keeping the two phases from 2PC, it reduces the overhead by sending messages in a circular fashion rather than back and forth, as illustrated in Figure V.1(b).

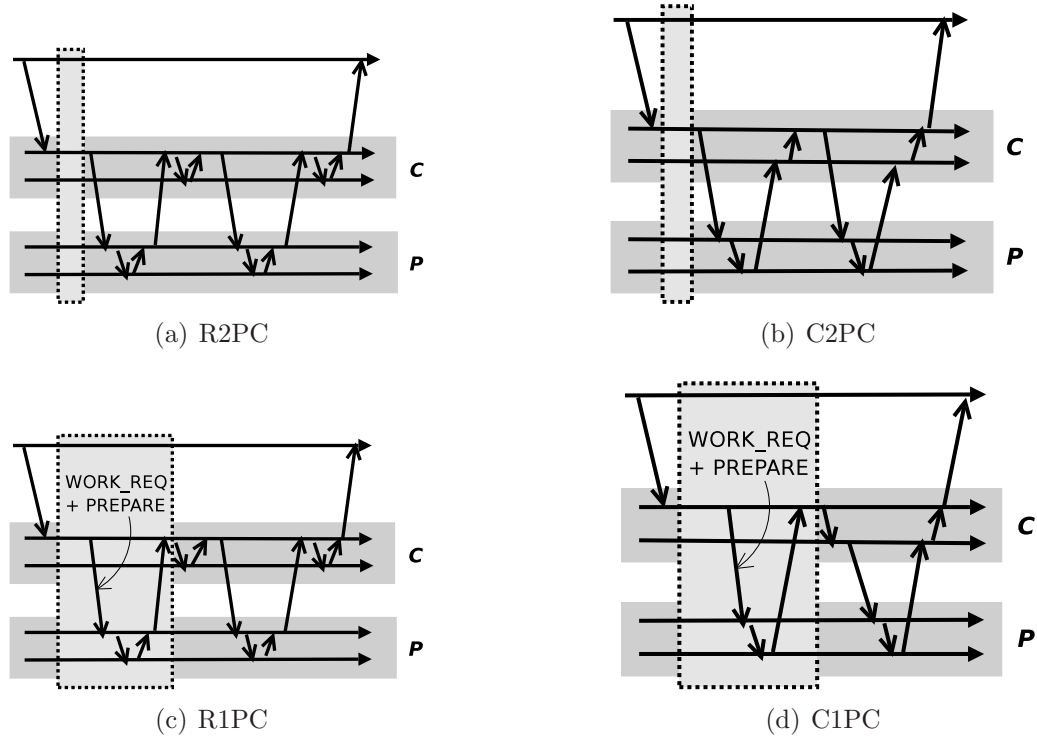


Figure V.1: Main-memory commit protocols for one backup

When a *backup participant* receives the vote from the corresponding *primary participant*, it replies to the *backup coordinator* instead. The *primary coordinator* has already voted, so the *backup coordinator* can make the decision to **ABORT** or **COMMIT** the transaction. The decision is then persistently stored by sending it to the *primary coordinator* which is responsible for sending it out to the *primary participants*. Again, the *backup participants* reply to the *backup coordinator*. And finally, the *primary coordinator* replies to the client.

### V.3.3 R1PC

A one-phased version of R2PC, the Replicated One-Phase Commit protocol (R1PC) [14] can be made by using the *Unsolicited Vote Protocol* (UV) [26]. When **DoWork** is sent to the participants, the **Prepare** request is piggy-backed on it, thus the first round is eliminated. It is executed as shown in Figure V.1(c), but the voting phase is included in the transaction processing. However, this impose some restrictions on the execution of the participants [1].

### V.3.4 C1PC

The *Circular One-Phase Commit protocol* (C1PC) [14] is a one-phased protocol based on UV and C2PC. Its execution is illustrated in Figure V.1(d). It differs from C2PC in that the *backup participants* reply to the *primary coordinator* instead of the *backup coordinator*. This is to allow additional work to be done by the primary after the participants have done their job. The primary makes the decision after receiving the votes and notifies the backup which forwards it to the participants.

## V.4 Extending to Multiple Backups

In some high-availability systems, having one backup may not be sufficiently fault tolerant to fulfill the system requirements. In these cases, two or even three backups may be needed. This section presents how to extend the protocols to multiple backups. During commit processing, at once the decision is made persistent, whether it is sent to backup or written to disk, the decision is final. Thus, despite failures, the outcome of the transaction is guaranteed. This allows the system to give an *Early Answer* [11] (EA) to the client. The multiple backup protocols is here presented with EA.

### V.4.1 R2PC

R2PC is easily extendible to multiple backups. It is always the primary which sends the messages to the backups and the backups always reply to the primary. Thus, the primary can send to all backups instead of just one, and make sure it has received replies from all before continuing the execution. Figure V.2(b) illustrates the failure-free execution of R2PC with two backups. As can be seen, the *primary participants* send out a **Prepare** message to the corresponding backups and wait to receive an acknowledgement for both of them, before they reply to the *primary coordinator*. It sends out a **Prepare** to both of its corresponding backups and waits to receive a reply from them before sending the decision to the client and *primary participants*. The second round is executed in the same manner.

### V.4.2 C2PC

A multiple backup version of a circular commit protocol is presented in Figure V.2(c). We keep calling this version C2PC, since it is just an extension of the C2PC protocol to multiple backups. It works by sending the **Prepare** message in a circle for each participant. That is, each *primary participant*

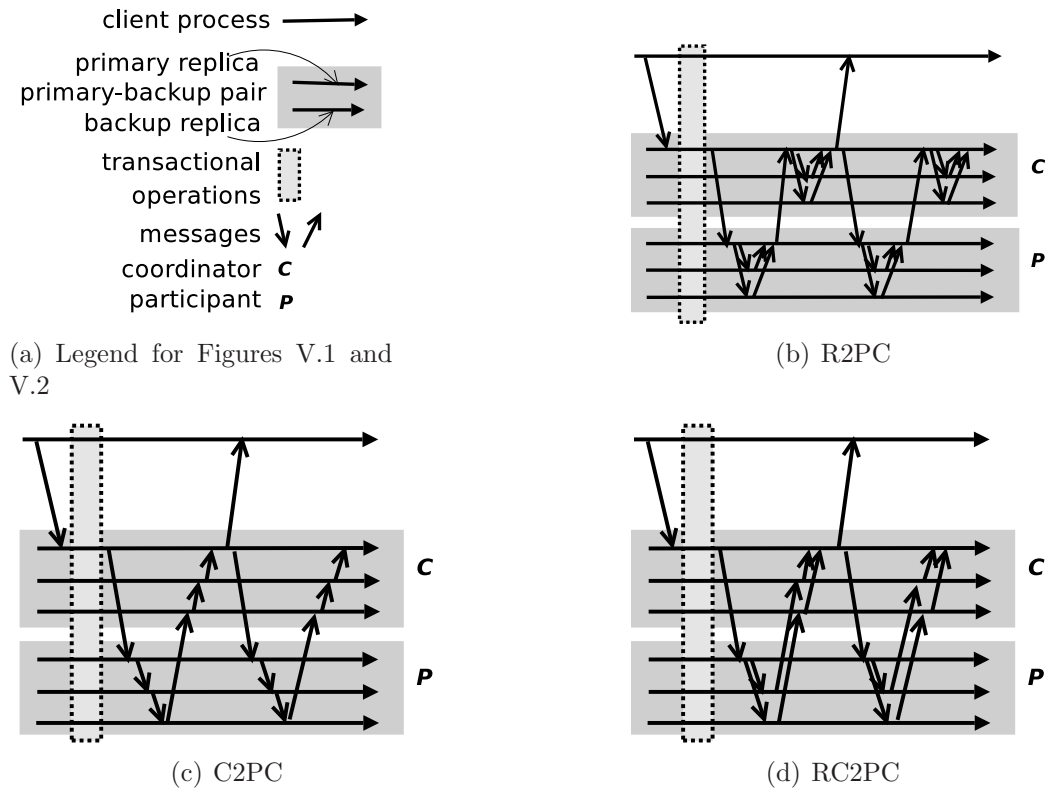


Figure V.2: Commit protocols for multiple backups, all executed with EA

sends `Prepare` to the first backup, which forwards it to the second, and so on, until all *backup participants* have seen the message. Then, it is sent to the last *backup coordinator*. It is responsible for collecting the votes, making a decision, and forwarding it to the next-to-last *backup coordinator*, and so on. When all *backup coordinators* have seen the decision, the *primary coordinator* replies to the client and sends the decision to the *primary participants*. As shown in Figure V.2(c), the second round is executed in a similar manner to the first one.

### V.4.3 RC2PC

The *Replicated Circular Two-Phase Commit protocol*, RC2PC, is a hybrid of C2PC and R2PC. As shown in Figure V.2(d), the *primary participant* sends `Prepare` to all *backup participants*. Each of them replies to the corresponding *backup coordinator*, which tallies the vote from the participants. Then, the *primary coordinator* makes a decision based on the votes received from the backups. It sends a reply to the client and the decision to the *primary participants*. As for R2PC and C2PC, the second round is executed as the

first one.

#### V.4.4 One-Phased Multiple Backups Protocols

One-phased commit protocols for multiple backups are easily achieved. This is done by including the first round of the commit processing in the transactional operators as for the single backup case. Thus, R2PC, C2PC, and RC2PC can be made into R1PC, C1PC, and RC1PC, respectively.

### V.5 The Simulation Model

To evaluate the performance of the commit protocols a discrete-event simulator was developed using Desmo-J [20]. The input parameters of the system is given in Table V.1. The model and the parameters have been more extensively explained in [15]. However, the cost of sending and receiving messages has changed. In line with update transactions, the messages sent over the network are assumed to be small, i.e. 500 bytes for the first message to reach each participant for each transaction, and 50 bytes for the others. Test experiments on a 100 Mbit/s Ethernet network show that no queuing effects for the network are likely to happen for the load of a distributed main-memory database. Also, small messages have a very short transmit time. Thus, the time to send a message from one node to another can be modelled as CPU execution time at the receiver and sender. The formulas for the response time are found by taking the average of 100.000 round-trip times for UDP datagrams with 10 bytes intervals. Measurements of CPU usages indicate that it takes twice as long to receive, as to send a message.

### V.6 Simulation Results

The simulation model in Section V.5 were used to perform simulations. Each of the general, multi-backup protocols in Section V.4 was executed. The results from executing simulations with the early-answer optimization are presented. First, the average response time for each of the protocols is presented, then the typical requirement of an upper bound on the maximum response time for 95% of the transactions [11] is explored. In Figures V.3 and V.5, the results for R2PC, CR2PC, and C2PC are marked as squares, triangles, and circles, respectively. Similarly, the results for R1PC, CR1PC, and C1PC in Figures V.4 and V.6 are marked as squares, triangles, and circles, respectively.

Table V.1: The input parameters of the simulation

<i>Parameter</i>	<i>Value</i>	
<i>Simulation Model</i>	Open, 20 nodes	
<i>Context Switch</i>	3.5 $\mu$ s, Timeslice: 5ms	
<i>Sim Capture Time</i>	160 s [40 - 200], 10 runs per result	
<i>Transactions</i>	3 single tuple update per txn	
<i>CPU Utilization</i>	Variable	
<i>Send Message</i>	$(26 + 0.055 * B) \mu$ s	
<i>Receive Message</i>	$(52 + 0.110 * B) \mu$ s	
<i>Long Operations</i>	700 $\mu$ s	DoWork, DoWorkAndPrepare
<i>Medium Operations</i>	150 $\mu$ s	BeginTxn, Prepare, Abort, Commit
<i>Short Operations</i>	50 $\mu$ s	WorkDone, Vote, Aborted, Committed

### V.6.1 Average Response Time

The time when the client sends a request until it receives an answer is the *response time* of the transaction. The average response time of the transactions was found for various utilizations up to 95%.

Figure V.3 shows the average response time when executing R2PC, CR2PC, and C2PC with the EA optimization. Using only one backup (see Figure V.3(a)), C2PC is better than R2PC. For two backups, as seen in Figure V.3(b), CR2PC is best for throughputs below 50 tps/node, while C2PC is best suited above that. R2PC is more effective than C2PC for throughputs below 15 tps/node. For the simulations with three backups (Figure V.3(c)), the same switches happen at around 37 and 7 tps/node.

The results for the one-phased protocols executed with EA are shown in Figure V.4. For one backup (see Figure V.3(a)), C1PC beats R1PC. When using two backups (Figure V.4(b)), the C1PC protocol is the best performer among these above 50 tps/node, while CR1PC is best below 50 tps/node. R1PC marginally beats C1PC under 3-4 tps/node. The results from simulating three backups indicate that the same changes happen at 40 and 10 tps/node.

As can be seen by comparing the plots in Figures V.3 and V.4, adding backups cause the performance to degrade. However, if more nodes are added, the total throughput does not need to decrease. The response time will, however, be slower with more backups compared to using fewer.

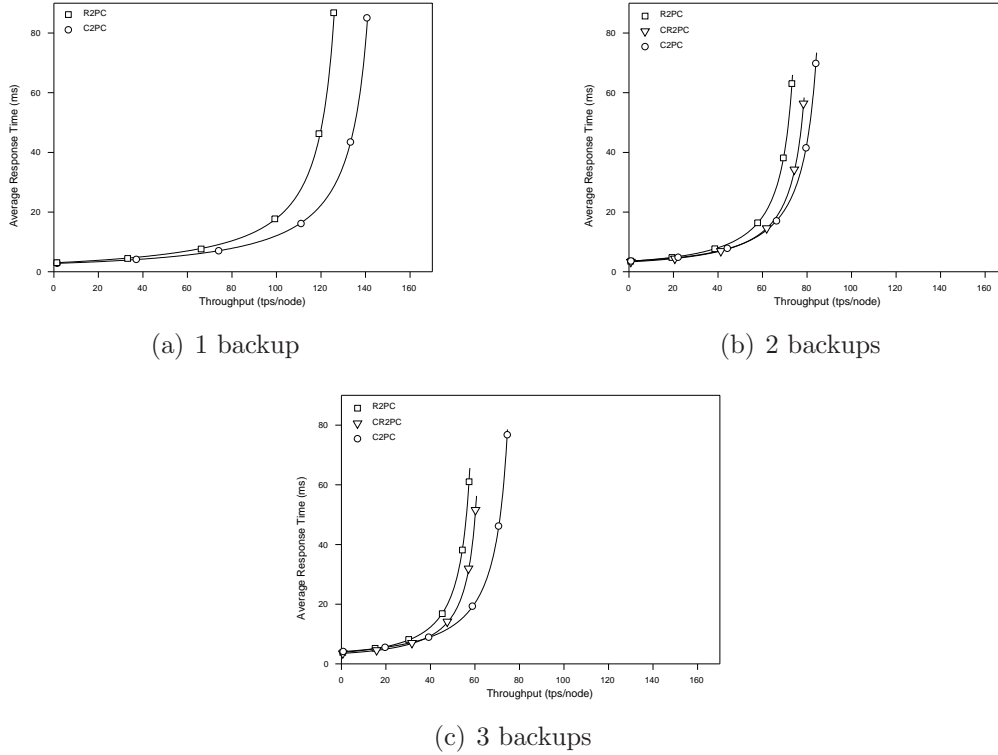


Figure V.3: Two-phased protocols executed with EA

### V.6.2 Response Time Demands

Typically real-time databases require most transactions to finish within a given time limit: for instance, response time of a maximum of 15 milliseconds for at least 95% of the transactions [11]. Figures V.5 and V.6 plot the results from these measurements for one, two and three backups using the EA optimization. The 5- and 10-millisecond limit is marked with dotted and dashed lines, respectively. The arrows mark where the plots for the simulation results cross the limit.

Figure V.5(a) shows the results for two-phased protocols using one backup. R2PC and C2PC tolerate 19 and 24 tps/node, respectively, before crossing the 5-milliseconds time limit. The 10-milliseconds time limit is passed at 53 and 64 tps/node, respectively. Executing the protocols with two backups (see Figure V.5(b)), R2PC, C2PC, and CR2PC tolerate 9, 11, and 14 tps/node, respectively, before crossing the 5-millisecond limit. If the limit is increased to 10 milliseconds, the same protocols tolerate 31, 37, and 38 tps/node, respectively. The results for the simulations with three backups are shown in Figure V.5(c). The same numbers for these simulations are 5, 4, and 8 for

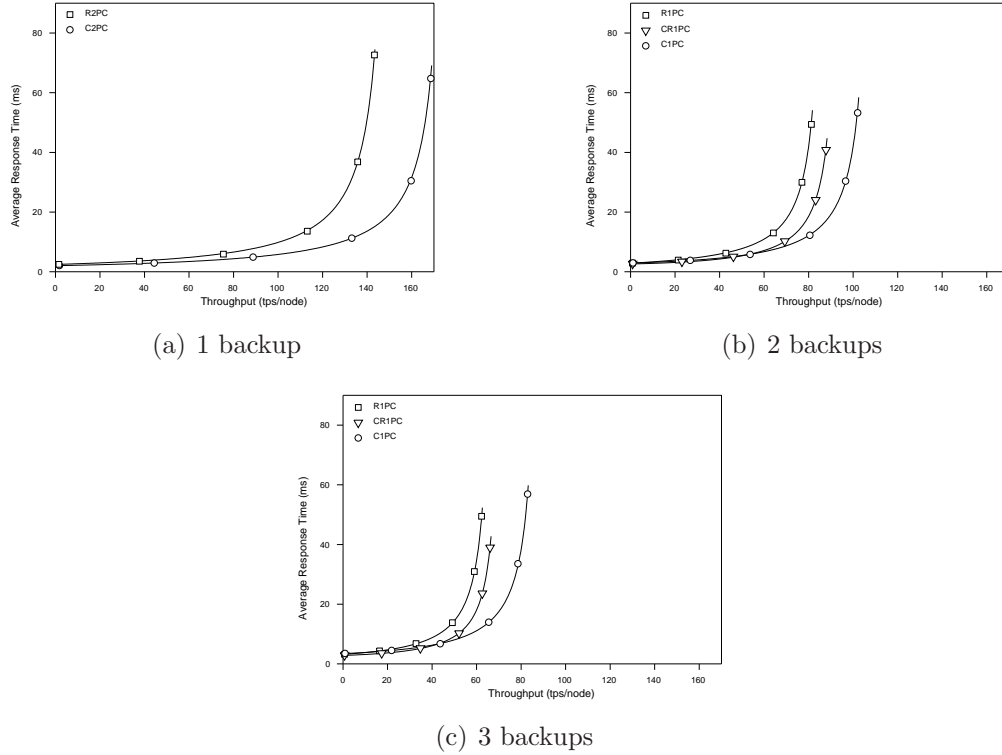


Figure V.4: One-phased protocols executed with EA

the 5-milliseconds limit and 23, 28, and 29 for the 10 milliseconds limit.

The simulation results for one-phased protocols are given in Figure V.6. Simulating one backup, the plots in Figure V.6(a) show that 34 and 50 tps/n-node can be executed using R2PC and C2PC, respectively, before crossing the 5-milliseconds limit. Using the 10-milliseconds limit, they tolerate 75 and 98 tps/node, respectively. For the two backups case in Figure V.6(b) and using the 5-milliseconds limit, R2PC, C2PC, and CR2PC tolerates 18, 24, and 26 tps/node, respectively. Using the 10-milliseconds limit, the same numbers are 42, 57, and 53. The 5-milliseconds limit when simulating three backups in Figure V.6(c) is 9, 14, and 18 tps/node for R2PC, C2PC and CR2PC, respectively. While 95% of the transactions finishes within 10 milliseconds, the maximum tps/node for the same protocols are 30, 43, and 40.

For the two-phased protocols, CR2PC tolerates the highest throughput before the time limits are crossed. R2PC is always the worst performer. Looking at the one-phased protocols, C1PC outperforms CR1PC if a 10-milliseconds limit is set. However, if a 5-milliseconds limit is used, CR1PC is the best protocol. For both limits, R1PC is the worst performer.

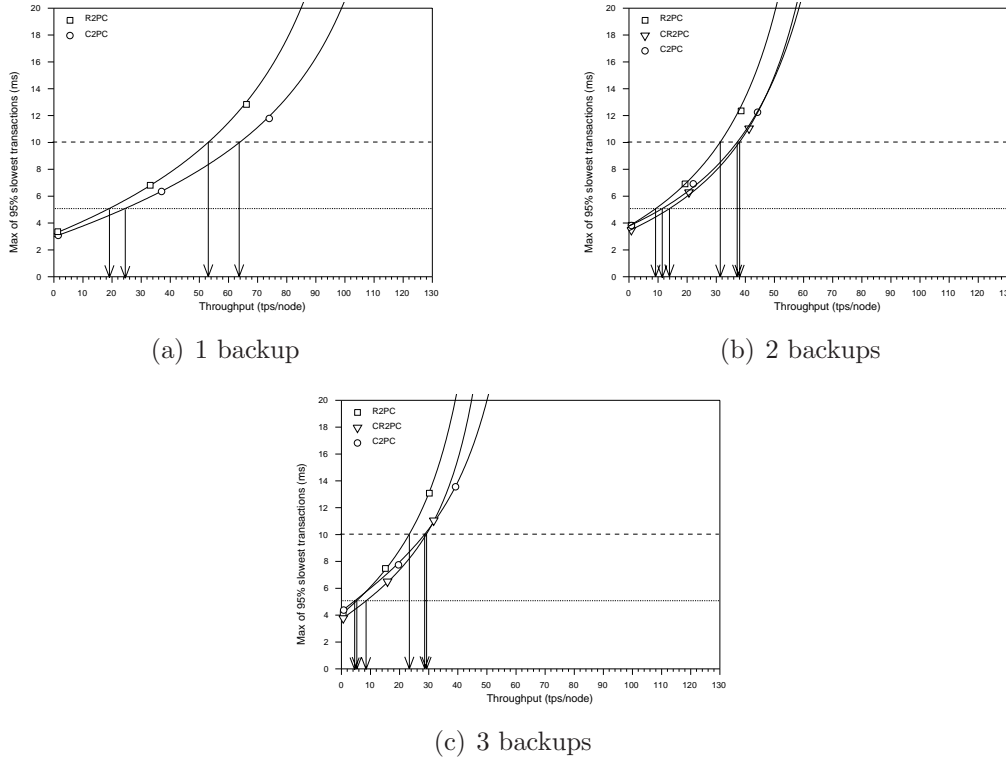


Figure V.5: Two-phased protocols using the EA optimization

## V.7 Analysis

The simulation results are verified by using the analytical model used in [15]. The model uses a multiple-class open-queueing network with exponentially distributed inter-arrival times. It assumes uniformity for all nodes, such that single-server queueing theory [12] can be applied. The steady-state solutions are derived using results from  $M/G/1$  queues [12]. Refer to [15] for more details.

Figure V.7 displays a comparison of the analysis and the simulation results for using three backups. The analysis are shown in black and the simulations in white. There are some small discrepancies between the results, but the overall shape of the analysis plots seem to confirm the simulations. The maximum difference is less than 10%, which is a reasonable difference. The comparisons of fewer nodes and the one-phased protocols show the same general behaviour, but are not shown because of space limitations.

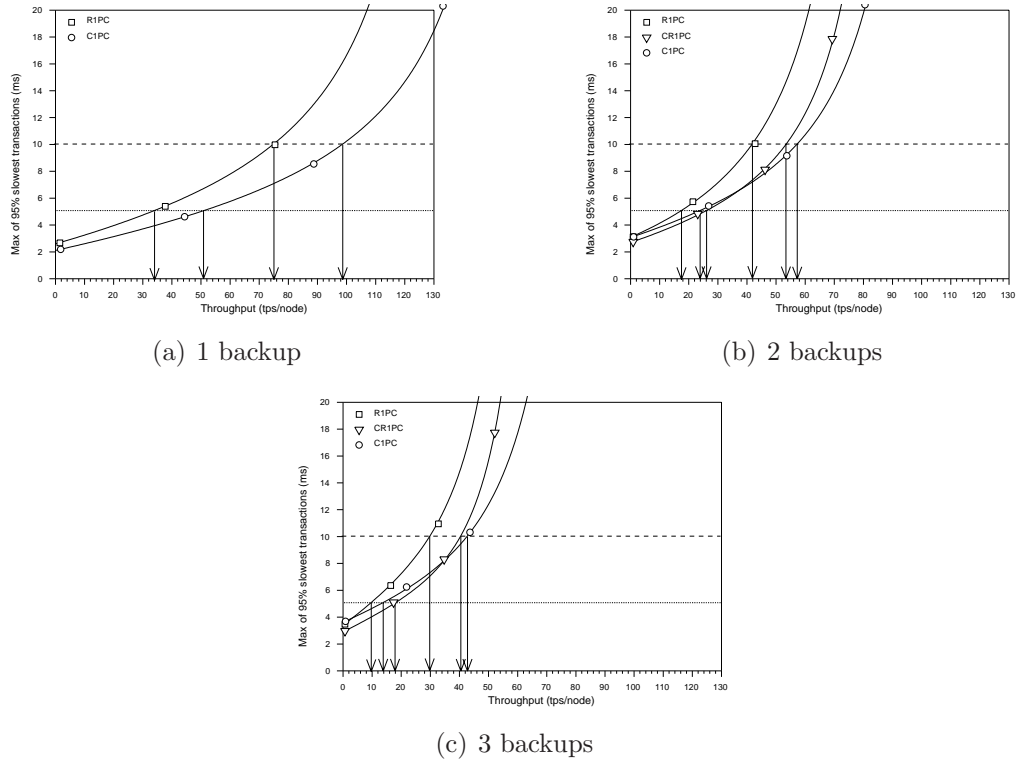


Figure V.6: One-phased protocols using the EA optimization

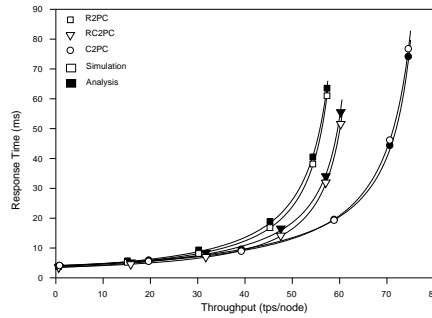


Figure V.7: Comparing analysis and simulation results for two-phased protocols and 3 backup nodes

## V.8 Evaluation and Conclusion

This paper proposes main-memory commit processing protocols for multiple backups. Using more than one backup improves the reliability and availability of the system, but as the simulations show, the performance suffers. This can be partially improved upon by adding more nodes to the system, such

that the load on the existing nodes are divided among multiple nodes.

CR1PC was found to be the best protocol for low and medium utilization. For high utilization, C1PC gives the best performance. R1PC should not be chosen as CR1PC or C1PC is better in all cases. If one-phased protocols cannot be used, CR2PC was found to be the best at low utilization, while C2PC is better for high utilizations. R2PC suffers from too much overhead, and is not a good choice.

If possible, the commit protocol should be chosen dynamically by letting the system change from CR1PC to C1PC under heavy load, and back to CR1PC at lighter load.

For future work, the protocols should be implemented in a real system to show the performance of the protocols for a real system. Also it would give insight in how the availability improves by adding more backups to the system. In addition, relaxing the consistency requirements of the systems, e.g. 1-safe design [10], may be tolerated for some application.

## References

- [1] Maha Abdallah, Rachid Guerraoui, and Philippe 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] Maha Abdallah and Philippe Pucheral. A single-phase non-blocking atomic commitment protocol. In *DEXA '98: Proceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 584–595, London, UK, 1998. Springer-Verlag.
- [3] Yousef J. Al-Houmaily and Panos K. Chrysanthis. 1-2PC: The one-two phase atomic commit protocol. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 684–691, New York, NY, USA, 2004. ACM Press.
- [4] Yousef J. Al-Houmaily and Panos K. Chrysanthis. ML-1-2PC: an adaptive multi-level atomic commit protocol. In *8th East-European Conference on Advances in Databases and Information Systems, ADBIS*, pages 275–290, September 2004.
- [5] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

- 
- [6] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, New York, NY, USA, 1984. ACM Press.
- [7] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 04(6):509–516, 1992.
- [8] Dieter Gawlick and David Kinkade. Varieties of concurrency control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985.
- [9] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.
- [10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [11] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, 1995.
- [12] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley & sons, 1991.
- [13] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Sergio Arévalo. A low-latency non-blocking commit service. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 93–107. Springer-Verlag, 2001.
- [14] Heine Kolltveit and Svein-Olaf Hvasshovd. The Circular Two-Phase Commit Protocol. In *Proceedings of International Conference of Database Systems for Advanced Applications*, pages 249–261. Springer-Verlag, 2007.
- [15] Heine Kolltveit and Svein-Olaf Hvasshovd. Performance of Main Memory Commit Protocols. Technical Report 06/2007, NTNU, IDI, 2007.
- [16] Heine Kolltveit and Svein-Olaf Hvasshovd. Main memory commit processing: The impact of priorities. *Proceedings of the International Conference of Database Systems for Advanced Applications*, 2008.

- 
- [17] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th Conference on Very Large Databases*. Morgan Kaufman, 1993.
- [18] Inseon Lee and Heon Young Yeom. A single phase distributed commit protocol for main memory database systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 44–51, Washington, DC, USA, 2002. IEEE Computer Society.
- [19] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R\* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.
- [20] Bernd Page and Wolfgang Kreutzer. *The Java Simulation Handbook. Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 2005.
- [21] Taesoon Park and Heon Y. Yeom. A consistent group commit protocol for distributed database systems. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS'99)*, 1999.
- [22] P. Krishna Reddy and Masaru Kitsuregawa. Blocking reduction in two-phase commit protocol with multiple backup sites. In *DNIS '00: Proceedings of the International Workshop on Databases in Networked Information Systems*, pages 200–215, London, UK, 2000. Springer-Verlag.
- [23] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 520–529, Washington, DC, USA, 1993. IEEE Computer Society.
- [24] James W. Stamos and Flaviu Cristian. A low-cost atomic commit protocol. In *Proceedings of the Ninth Symposium of Reliable Distributed Systems*, 1990.
- [25] James W. Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4), 1993.
- [26] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.



# Paper VI

Efficient Execution of Small (Single-Tuple) Transactions in  
Main-Memory Databases.

Heine Kolltveit and Svein-Olaf Hvasshovd.

In *Advances in Databases and Information Systems*,  
SpringerLink.

September 5 - 9, 2008, Pori, Finland



# Efficient Execution of Small (Single-Tuple) Transactions in Main-Memory Databases

Heine Kolltveit and Svein-Olaf Hvasshovd

Department of Computer and Information Science  
Norwegian University of Science and Technology  
NO-7491 Trondheim, Norway

## Abstract

Various applications impose different transactional loads on databases. For example for telecommunication systems, online games, sensor networks, and trading systems, most of the database load consists of single-tuple reads and single-tuple writes. In this paper, approaches to handle these single-tuple transactions in main-memory systems are presented. The protocols are evaluated by simulation and verified by statistical analysis. The results show that 70 - 150% more transactions can be executed while keeping response times low using the new approach, compared to a state-of-the-art protocol.

## VI.1 Introduction

In recent years, there has been a gigantic increase in subscribers of mobile services. From late 2005 to November 2007, there was an increase from 2.14 billion to 3.3 billion subscriptions worldwide [21, 26]. This is a staggering 50% increase in less than two years. In addition, prices for usage keeps dropping, increasing the usage by every subscriber [28]. Telecommunication (telecom) systems must facilitate the increasing load. For instance, the *Home Location Registry* is the central database in any mobile network. Its most important task is to manage subscriber mobility. Each time a mobile subscriber is called, the location of the subscriber is read, and each time a subscriber moves from one *Location Area* to another, the location of the subscriber is updated. Consequently, most of the load are single-record writes and single-record reads.

The most important requirements for telecom databases are [27]:

**Reliability:** No more than 30 seconds downtime per year, corresponding to a  $10^{-6}$  probability of being out of service. This means no planned system downtime. Also, disaster failures such as earthquakes or fires must be masked.

**Performance:** Up to 10-20,000 transactions per second. Most of these are either read or write a single record.

**Delays:** Both write and read operations must have delays ranging from 5 to 15 milliseconds. It is also generally required that a certain percentage (e.g. 95%) of transactions finish within the time limit [10].

As a consequence of the delay requirement, and because disk accesses are slow, transactions cannot read from or write to disk. In contrast, fast main-memory accesses [5] should be used instead. However, since main memory is *volatile*, the reliability requirement cannot be met unless data is *replicated* and kept at multiple nodes.

Normal SQL queries have multiple interactions with the client. For instance, a user making an online reservation will first issue a search, then reserve one or more items, before finally completing the transaction. In contrast, the common single-tuple accessing transactions in telecom system can be implemented as canned transaction or Persistent Stored Modules (PSM) [11]. A PSM is a precompiled transaction which is stored in the database management system, and the client simply invokes it with input parameters and the result is returned. PSM makes it easy to distinguish the transaction type. Since the simple transactions are the predominant transactional load on the telecom database, their execution should be optimized.

This work uses the fact that the type and size of transactions may be known at execution time for PSMs. Therefore, the semantic differences between read-only transactions (queries) and write-only transactions (updates) can be used to reduce the delay of these transactions and improve the performance of the system, without affecting its overall reliability.

The motivation of this work is the high load on transactional systems with real-time requirements as the ones presented for telecom systems above. The main contributions of this work are main-memory transaction execution and commit protocols where the differences between single-record updates and single-record queries are exploited to increase the throughput and reduce the delay. The results of this work can be favorably used in telecom and other systems with similar loads and requirements for reliability, performance, and delays. Examples include online games, trading systems and sensor networks.

The rest of the paper is organized as follows. Section VI.2 gives an overview of related work. Section VI.3 presents the properties of the two

types of transactions, and how to handle them, while Section VI.4 proves the correctness. Section VI.5 outlines the simulation model and parameters used for the simulations in Section VI.6. The simulations are compared to a statistical analysis in Section VI.7, while Section VI.8 gives some concluding remarks.

## VI.2 Related Work

This section presents important related work. First, approaches using the transaction type are presented. Then, the most relevant main-memory commit processing approaches are introduced.

### VI.2.1 Transaction Type

The idea of using the type of transaction to optimize transaction processing was proposed several decades ago. It uses the fact that the system can treat various transactions types in different ways. For instance, not all transactions require synchronization as strong as global locking. In [2], four synchronization levels are defined and protocols are proposed.

Multi-versioning is suggested to be used for long read-only transactions [4, 20]. A read-only transaction can then use a consistent versioned copy of the database by reading old snapshot data. In this way, the interference between long read-only transactions and shorter and more urgent update transactions is minimized. However, it is not suitable for all applications since data may be outdated. A special case of read-only transactions are *free queries* [6]. Free queries have no consistency or concurrency requirements, and can thus be executed without locking. Assuming that update transactions do not write to the database until commit time, an example of a transaction which is always a free query is one which reads only one record. Such a transaction will always be consistent [6].

In [1], an approach is presented where multi-versioning is used to execute write-only operations logically in the future. This is the opposite of the versioned read-only optimization. Standard write transactions read the value, adjust it, and then write it back. These are called *read-write* transactions. *Write-only* transactions [9, 1], however, do not read the value before updating it. By distinguishing between these types of transactions and using version control, the interference between read-write and write-only transaction are minimized.

The read-only commit optimization [7] is based on the observation that read-only operations have no work to do during the first phase of the two-

phase commit protocol. Hence, it can be skipped. The second phase must still be executed since the locks must be released. This optimization has little effect on one-phased protocols, since the first phase is included in the transaction execution.

Ramamritham [25] divides the transaction processing load for real-time systems into three types of transactions:

1. Read-only transactions, which read data from a database and do not make any updates.
2. Write-only transactions, which write external data into the database.
3. Read-write transactions, which read data from a database, update it, and write it back.

The RODAIN database architecture [23, 18] is designed to be used in telecommunication systems. It is a real-time object-oriented architecture of a database management system. In [23], it is argued that a telecommunication system should distinguish between three transaction types: (1) Short and frequent simple-read transactions, (2) long and frequent update transactions and (3) long and rare read and write transactions. These correspond to the three transaction types by Ramamritham presented above. However, the large volume of short simple-update transactions generated by the mobility of the subscribers seem to be overlooked, and although distribution of data across a cluster is mentioned, the discussion is only for a single primary-backup pair. How to do global commit is not discussed. Also, the approach relies on the disk at the backup node. In addition, if the primary asks for the acknowledgement again (i.e. the message has disappeared), the backup may have deleted the transaction from the transaction table.

### VI.2.2 Main-Memory Commit Processing

This section introduces some existing approaches from previous work which are used in this paper. The first is the *Circular One-Phase Commit protocol* (C1PC) [13]. Figure VI.1(a) shows the failure-free execution of C1PC. When receiving a transaction request, the coordinator piggybacks a **Prepare** message on every **DoWork** to each of the participants. This instructs the participants to prepare the transaction and vote. Using C1PC, the primary participant sends the necessary log records to the backup participant, which sends its vote to the primary coordinator. The primary coordinator collects the vote and decides the outcome of the transaction. The outcome is sent to the backup coordinator which gives an *Early Answer* [10] to the client

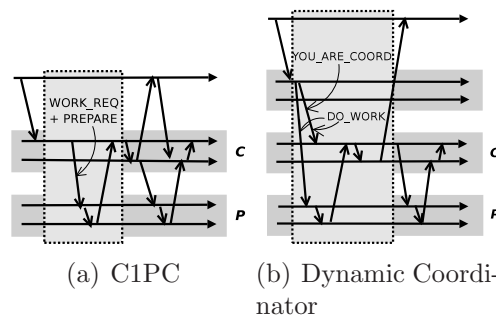


Figure VI.1: Existing commit processing approaches and optimizations for main memory

and notifies the primary participants. The decision is then forwarded to the backup participants. When the backup coordinator has received acknowledgements from all participants, it notifies the primary coordinator, and the transaction is completed.

Typically, the node where the client request arrives becomes the transaction coordinator. As each transaction arrives at a random node in the system, there is no guarantee that the coordinator is placed at a node with data accessed by the transaction. The dynamic coordinator approach was presented in [15]. In this approach, the transactional workload is split by the original coordinator, and sent to each participant (node holding data to be accessed). One of the participants are chosen as the new coordinator and a `YouAreCoordinator` flag is added to the message. The message to the other participants contains a `CoordinatorIs` variable which identifies the new coordinator. In this way, the number of messages and processing required during commit processing are reduced, since the number of participants are reduced by one.

Piggybacking and prioritizing can also be used. In [16], an approach where messages and tasks are prioritized depending on whether they are urgent or not is presented. A task is urgent if it is in the critical path before the early answer is sent to the client. Non-urgent tasks are only executed when there are no urgent tasks to do, and non-urgent messages are not sent until there are no urgent messages. To avoid starvation, however, messages and tasks which have waited longer than a given time limit are executed. Also, when a message is sent to another node, both the outgoing urgent message and non-urgent message queues are checked to see if there are any other messages going to the same destination. If there are, they are *piggybacked* on the first message. Because of the reduced overhead of sending fewer messages, bandwidth and time is saved. The prioritizing reduce the delay observed by the client, since urgent tasks and messages do not have to wait for the

non-urgent, while the piggybacking improves the throughput. Both of these have more effect if the utilization is high (70% and more) [16].

## VI.3 Using The Semantics of Read and Write Operations

This work takes advantage of the different semantics of read and write operations. For the purpose of clarifying the semantic differences, transactions are classified into three groups in this paper:

**Simple update** The transaction updates a single record.

**Simple query** The transaction reads a single record.

**General transaction** The transaction reads and/or writes two or more records.

The general transaction is efficiently handled using C1PC with Early Answer, Dynamic Coordinator, and prioritizing and piggybacking optimizations presented in Section VI.2. This leaves the two simple transaction types to be presented in this section.

### VI.3.1 Simple Update

A simple update transaction can be treated differently than the general transaction. In particular, it does not need to execute a distributed commit protocol. For disk-based databases, it is enough to set a write lock, update the record, force the log to disk, and return an answer to the client. Thus, a distributed commit protocol is not required. In contrast, main-memory databases persistently store the log by sending it to the backup. An answer can be returned to the client after both primary and backup know the outcome of the transaction.

When a simple update transaction arrives at a node in a main-memory system, the data tables are checked to see if the record being updated resides at the current node. If not, the transaction is forwarded to the proper node. This is the same as the dynamic coordinator optimization presented in Section VI.2 with the number of participants set to one.

At the proper node, the transaction must acquire a write lock on the record and perform the write. Then the log is sent to the backup. Piggybacked on the log are `Prepare` and `YouAreCoordinator`. Thus, it is a version of the *transfer of commit* optimization [7]. The backup becomes the coordinator

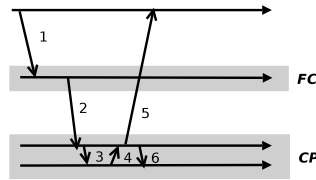


Figure VI.2: The protocol for single-update

```

simple update:                                1
if (data at another node) forward txn;
else {
  acquire write lock on record;                4
  write record;
  if (successful) {
    send log + prepare + root to backup;        7
    wait for outcome from backup;
    decide outcome;
    return early answer to client;             10
    release lock;
    send acknowledgement to backup;
  } else {
    decide abort;
    release lock;
    return error code to client;               16
  }
  forget transaction;
}                                              19
    
```

Listing VI.1: Primary coordinator

```

simple update:                                20
save log;
if (ok) outcome = COMMIT;
else outcome = ABORT;                          23
decide outcome;
send outcome to primary;
wait for acknowledgement;                       26
forget transaction;
    
```

Listing VI.2: Backup coordinator

and, since it knows that the primary is already prepared, it can decide to COMMIT or ABORT the transaction depending on its own vote.

To guarantee that the system is *recoverable*, the COMMIT log record must be persistently stored, before a reply can be given to the client. ABORTS does not need this since a missing decision log record is presumed to be an ABORT [7]). The COMMIT is stored persistently by sending it to the primary. Since both the primary and backup now know the outcome, the primary can send an *Early Answer* to the client and an acknowledgement (**Committed**) to the backup. Then, the backup can forget about the transaction, i.e. remove it from the Active Transaction Table. After receiving an acknowledgement, the primary can also forget about the transaction.

Listings VI.1 and VI.2 give the algorithms for the primary and backup, respectively.

### VI.3.2 Simple Query

Simple query transactions require even less work than the simple update transactions. A read does not need to be logged and, since only a single

simple read:	
<b>if</b> (data at another node) forward txn;	
<b>else</b> {	29
acquire read lock;	
read record;	
release lock;	32
<b>return</b> answer to client;	
forget transaction;	
}	35

Listing VI.3: Primary coordinator

record is read, no commit coordination is required. However, to avoid reading uncommitted data (a record written by an uncommitted transaction), a read lock must be acquired before the record is read.

The algorithm is given in Listing VI.3. As for simple updates, it makes use of the dynamic coordinator optimization, forwarding the transaction to the proper node. It waits for the read lock to be acquired, reads the records, and releases the lock. Then, an answer can be sent to the client and the transaction can be forgotten.

## VI.4 Correctness

This section proves the correctness of the Simple Update (SU) protocol. Each of the properties given in Section VI.4.1 is proved in Section VI.4.2 in this order: **NB-AC2**, **NB-AC3**, **NB-AC4**, **NB-AC1** and **NB-AC5**. The Simple Query (SR) protocol does not need a distributed commit protocol and is therefore trivial to prove.

### VI.4.1 The Non-Blocking Atomic Commitment Problem

An atomic commitment protocol ensures that the participants in a transaction agree on the outcome, i.e. ABORT or COMMIT. Each participant votes, YES or NO, on whether it can guarantee the local ACID properties of the transaction. All participants have the right to *veto* the transaction, thus causing it to abort. The *Non-Blocking Atomic Commitment* problem, NB-AC, has these properties [3, 8]:

**NB-AC1** *<uniform agreement>* All processes that decide reach the same decision.

**NB-AC2** *<integrity>* A process cannot reverse its decision after it has reached one.

**NB-AC3** *<uniform validity>* COMMIT can only be reached if *all* processes voted YES.

**NB-AC4** *<non-triviality>* If there are no failures and no processes voted NO, then the decision will be to COMMIT.

**NB-AC5** *<termination>* Every correct process eventually decides.

### VI.4.2 Proof

**Lemma 1. NB-AC2:** *A process cannot reverse its decision after it has reached one.*

*Proof.* The algorithm for the backup use an if-else statement to decide only once. The primary accepts the outcome decided by the backup. ■

**Lemma 2. NB-AC3:** *The COMMIT decision can only be reached if all processes voted YES.*

*Proof.* The backup decides COMMIT if the primary is successful (voted YES) and itself is successful. The primary can only decide commit if the backup has decided commit. Thus, all processes have voted YES for the COMMIT decision to be reached. ■

**Lemma 3. NB-AC4:** *If no process failed and no process voted NO, then the decision will be COMMIT.*

*Proof.* If no process failed, and no process voted NO, then, since the communication system is reliable, the backup receives YES from the primary and itself. Thus, COMMIT is reached (line 22). ■

**Lemma 4. NB-AC1:** *All processes that decide reach the same decision.*

*Proof.* The primary can decide ABORT if it is not successful in writing the record. The backup does not need to be informed of the decision, since it does not even know about the transaction. Aside from that, the primary can only decide ABORT if the backup decided ABORT first. Similarly, the primary can only decide COMMIT if the backup decided COMMIT first. As proven in Lemma 2, COMMIT can be decided (line 22) only if the primary voted YES. A vote YES from the primary implies that it was successful at updating the record and cannot decide ABORT. Hence, the primary and backup cannot reach diverging decisions. ■

**Lemma 5. NB-AC5:** *Every correct process eventually decides.*

*Proof.* To enable a process to decide in the presence of failures, all failure scenarios as well as the scenario with no failures must be handled. These scenarios can occur:

- 1: The primary fails before sending the vote to the backup or an error code to the client.
- 2: The primary fails after sending the reply to the client, but before sending an `AckMsg` to the backup.
- 3: The primary fails after sending the `AckMsg` to the backup.
- 4: The backup fails before sending the decision to the primary.
- 5: The backup fails after sending the decision to the primary.
- 6: No node fails.

Scenario (1): The backup does not know of the transaction and is consistent. The client can later resend the request.

Scenario (2): When the backup does not receive an acknowledgement from the primary, it can complete the transaction with the decided outcome. If the primary successfully sent a reply to the backup, the client may receive duplicate replies which must be filtered.

Scenario (3): This does not affect the processing of the current transaction.

Scenario (4): If the backup node has failed, the primary can safely abort the transaction.

Scenario (5): The transaction is completed by the primary.

Scenario (6): This is proven similarly to Lemma 3. Since no process failed and the communication system is reliable, either the primary decides ABORT or the backup receives a YES vote. If the backup receives a YES, it decides either COMMIT in line 22 or ABORT in line 23. If the primary decides ABORT, the backup will not decide. However, since an abort leaves the system in the same state as before the transaction arrived, the state of the backup will be identical to an aborted state.

All scenarios are handled, thus, all correct processes eventually decide. ■

**Theorem 2.** *SU is a valid non-blocking atomic commitment protocol.*

*Proof.* Since SU satisfies properties **NB-AC1** - **NB-AC5** it solves NB-AC. ■

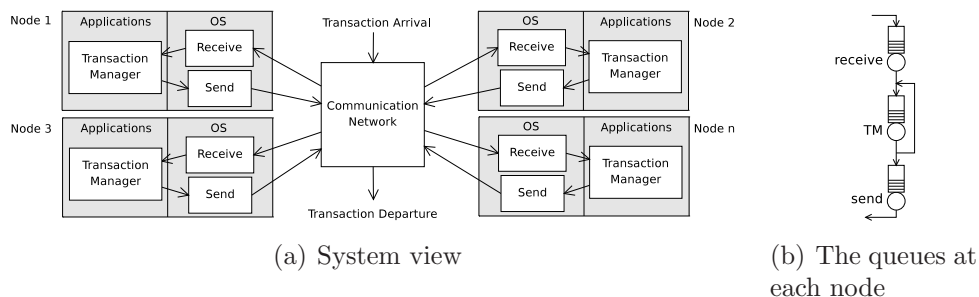


Figure VI.3: Logical model of the system

## VI.5 The Simulation Model

To evaluate the performance of various main-memory commit protocols, a simulator of a realistic distributed transaction processing system has been developed. It has been implemented using Desmo-J, a framework for discrete-event modelling and simulation [24].

Measurements to provide the input values for the simulations are performed on an AMD Athlon(TM) 64 bits 3800+ processor, running a Linux 2.6 kernel. The length of the transaction operations are derived from measurements performed on a Java main-memory database prototype developed by Løland and Hvasshovd [19].

The distributed transaction processing TP system considered in this paper is composed of  $N$  nodes connected through a communication network, as shown in Fig. VI.3(a). Each node has a *transaction manager* (TM) which is responsible for orchestrating the correct *local* execution of transactions. The correct *global* execution of transactions is ensured by coordination between TMs.

Each transaction is associated with a *coordinator*, which is responsible for execution and termination. Typically, the coordinator is the *transaction manager* (TM) at the node where the transaction was *initiated*. After the transaction execution is finished it is terminated using an atomic commitment protocol between the coordinator and participant TMs.

The distribution of processing and data is transparent to the transaction clients. The coordinator is responsible for issuing subtransactions to the appropriate nodes. Nodes receiving subtransactions are called *participants*. Generally, subtransactions can create further subtransactions, causing a multi-level transaction execution tree [22]. However, this paper discusses only trees that are one level deep. Transactions are assumed to be precompiled, and all nodes know where data are located. Thus, the coordinator can forward subtransactions directly to the correct participant.

Table VI.1: The input parameters of the simulation

Parameter	Value
Simulation Model	Open
Context Switch	3.5 $\mu$ s
Timeslice	5 ms
Send Message	(26+0.055*B) $\mu$ s if B<1500, (73+0.030*B) $\mu$ s else
Receive Message	(52+0.110*B) $\mu$ s if B<1500, (146+0.060*B) $\mu$ s else
Long Operations	700 $\mu$ s   DoWork,DoWorkAndPrepare
Medium Operations	150 $\mu$ s   BeginTxn,Prepare,Abort,Commit
Short Operations	50 $\mu$ s   WorkDone,Vote,Aborted,Committed
Simulation Time	200 s
Capture Time	160 s [40 - 200]
Simulation runs	10
Subtransactions	Variable (1 or 3)
Transactions	Single-tuple update or read
# of simulated nodes	20
CPU Utilization	Variable

We model each node to consist of a *transaction manager* (TM) executing on top of the operating system. The *operating system* (OS) is responsible for receiving and sending messages, while the transaction manager processes the requests. Other applications are assumed to be invoked from the TM. For each node, there are three First-In-First-Out queues: One for incoming messages, one for outgoing messages, and one for tasks to be processed by the TM. This is shown in Fig. VI.3(b). A processed task can result in a new local task, which is inserted at the end of the TM-queue, or a remote task, which is inserted into a message and sent by the operating system.

Process *context switches* between the two processes of the system, OS and TM, are modelled. The OS alternates between sending and receiving messages. Measurements show that these are performed in just 3.5  $\mu$ s. The *timeslice* is the maximum time given to each process to execute requests before another process is given time at the CPU. The default time slice for Linux kernel 2.6 systems is 100 ms and the minimum is 5 ms. To avoid delaying operations too long for this application, a time slice of 5 ms is chosen.

An *open* simulation model is used. The utilization of the system was varied from 1% - 97.5% to see the impact on the system performance. To

be able to set the utilization for each protocol and optimization, a single transaction was run for each combination, and the total service demand of it was found. Then, since there are only a few context switches per transaction at high utilization, the costs of nearly all context switches was subtracted from the total. Then, the altered service demand was used as a divisor to find the maximum throughput for each of the protocols per node.

The transaction load is composed of simple reads, simple updates and standard transactions. The standard transaction is chosen to include three participants, and the operations can be both read and write. The ratio between the transactions are 20 simple reads and 20 simple writes for every standard transaction. The coordinator and participants are chosen randomly and uniquely for each transaction. Since none of the 3 records chosen is assumed to be at the same node, each of the three subtransactions of a transaction has a unique destination node.

In line with update transactions, the messages sent over the network are assumed to be small, i.e., 500 bytes for the first message to reach each participant for each transaction and 50 bytes for the others. Test experiments on a 100 Mbit/s Ethernet network show that no queuing effects for the network are likely to happen for the load of a distributed main-memory database. Also, small messages have a very short transmit time. Thus, the time to send a message from one node to another can be modelled as CPU execution time at the receiver and sender. The formulas for the response time are found by taking the average of 100.000 round-trip times for UDP datagrams with 10 bytes intervals. Measurements of CPU usages indicate that it takes twice as long to receive, as to send a message.

Operations are divided into three groups: Long, medium and short. These are made to reflect the various service demands needed by an *average* operation of that kind. Long operations (`DoWork` and `DoWorkAndPrepare`) take 700  $\mu$ s, medium operations (`BeginTxn`, `Prepare`, `Abort` and `Commit`) take 150  $\mu$ s and short operations (`WorkDone`, `Vote`, `Aborted` and `Committed`) takes 50  $\mu$ s.

As the primary focus of this paper is the performance of commit protocols, the database internals are not modelled. A very large number of records is assumed, giving negligible data contention, hence, lock waiting effects have been ignored.

Each simulation lasted for 200 simulated seconds. The first 40 seconds are disregarded, as we are interested in the *steady-state* system. For each result presented, 10 simulation runs with random seed generators were made.

We assume the typical transaction load of 80% read-only transactions and 20% write-only transactions [17], and vice versa for comparison.

Table VI.1 summarizes the simulation parameters.

## VI.6 Simulation Experiments

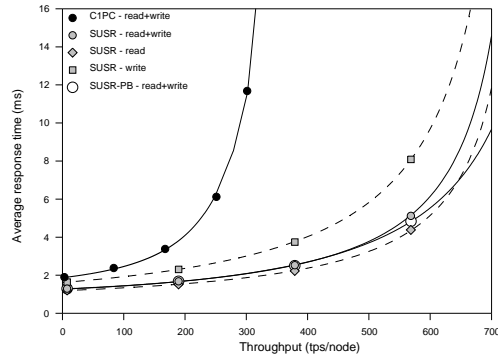
The simulation results were gathered using the system model presented in Section VI.5. The algorithms presented in Section VI.3 (here referred to as Single Update - Single Read, SUSR) were compared with an efficient commit processing protocol (C1PC) and not using the dynamic coordinator optimization. In this section, the simulation results are presented. First, graphs of the average response time versus throughputs are presented, then the graphs for the upper bounds on the maximum response time for 95 % of the transactions [10] are explored.

### VI.6.1 Average Response Time

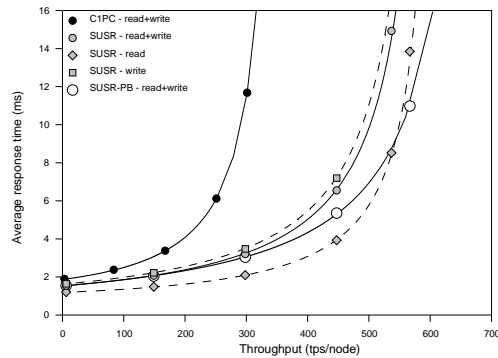
Simulations were performed for C1PC, SUSR, and SUSR with piggybacking (SUSR-PB) while varying the load on the system. The average response time versus throughput is shown in Figure VI.4. C1PC is shown in black, SUSR in grey, and SUSR-PB in white. For SUSR, the average response time for read, write, and combined read and write operations are shown as diamonds, squares, and circles, respectively.

Figure VI.4(a) shows the results where 80% of the load consist of simple reads and 20% simple updates. C1PC has the longest average response time and the least throughput. For low utilization, SUSR and SUSR-PB have on average two-thirds of the response time of C1PC. Since SUSR and SUSR-PB support higher throughput than C1PC, a higher utilization makes the relative improvement higher. Up to approximately 500 transactions per second per node, SUSR and SUSR-PB has the same response time. For higher utilization, piggybacking and priority of operations reduce the response time. For SUSR, the read-only transaction part of the load has an average response time of 25% less than the write transactions at low utilization. At 80% utilization, the difference has risen to approximately 50%. Since the load mostly consists of read transactions, the average for both read and write transactions are closer to the read average than the write average. At very high throughput, the average response time for SUSR-PB gets shorter than the same for the read transaction part of SUSR, because of the piggybacking and prioritizing optimizations.

Comparing the results in Figure VI.4(b) to the previously discussed results presented in Figure VI.4(a), the results for C1PC are, as expected, equal. This is because C1PC treats reads and writes in the same way. For SUSR and SUSR-PB the results are similar. The two major differences are that the maximum possible throughput is lower for the former, and the combined read and write transaction results for SUSR are closer to the average



(a) 80% simple reads and 20% simple updates



(b) 20% simple reads and 80% simple updates

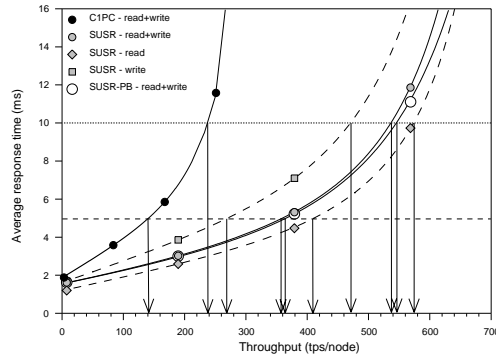
Figure VI.4: The average response time.

write response time. These are both caused by a larger part of the transactional load consisting of slower write transactions. For 80% update transactions, the piggybacking and priority improvements show a difference between SUSR and SUSR-PB already at 250 transactions per second per node. This is because there are more messages to be piggybacked for update transactions than for write transactions.

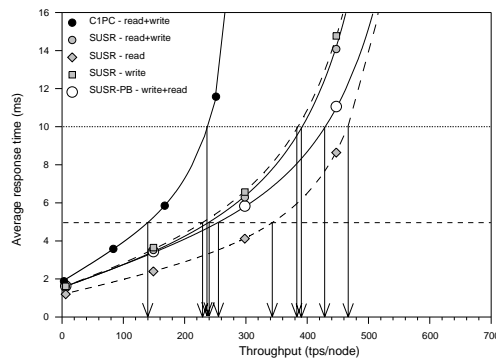
## VI.6.2 Response Time Demands

To examine the typical delay requirement for a certain percentage (i.e. 90%, 95%, 97%, 98%, or 99%) of transactions to finish within a given timelimit, Figure VI.5 plots the maximum response time for the 95% shortest transaction response times. The 5- and 10-milliseconds time limit is shown in dashed and dotted lines, respectively. The arrows points at the maximum throughput for each of the protocols at each time limit.

Figure VI.5(a) shows the results from executing the protocols with a load



(a) 80% simple reads and 20% simple updates



(b) 20% simple reads and 80% simple updates

Figure VI.5: The maximum response time for the 95% quickest transactions of 80% read transactions and 20% write transactions. For a time limit of 5 milliseconds, C1PC supports 140 tps/node and SUSR and SUSR-PB 360 tps/node. For a time limit of 10 milliseconds, C1PC tolerates 240 tps/node, while SUSR and SUSR-PB support 540 and 550 tps/node, respectively. For SUSR, data from the read-only and write-only transactions are plotted separately (grey diamonds and grey squares, respectively). If, for instance, the requirement for read-only transactions is that 95% complete in 5 milliseconds, while write-only can tolerate delays of up to 10 milliseconds, the maximum possible throughput is 410 tps/node (the minimum of the 5 millisecond limit for read-only and the 10 millisecond limit for write-only).

The results from executing the protocols with 20% read-only transactions and 80% write-only transactions are plotted in Figure VI.5(b). These show that C1PC tolerates 140 and 240 tps/node for the 5 and 10 milliseconds limits, respectively. SUSR and SUSR-PB support up to 240 and 260 tps/node, respectively, while keeping 95% of transaction response times below 5 milliseconds. If the limit is 10 milliseconds, the numbers are 390 and 430

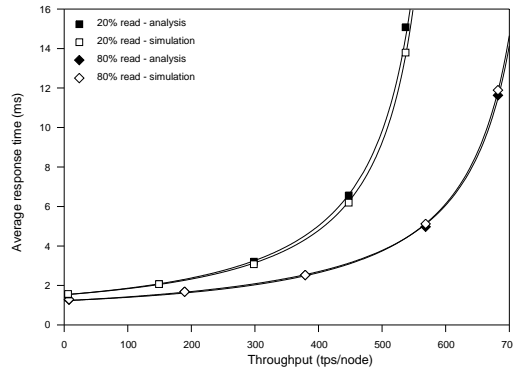


Figure VI.6: Comparison between analyzes and simulations of SUSR

tps/node for SUSR and SUSR-PB, respectively. As for the read-heavy load, the SUSR read-only and write-only transactions are plotted separately, and if 95% of the read-only transactions and write-only transactions must complete in 5 and 10 milliseconds, respectively, the maximum throughput is 340 tps/node.

Looking at SUSR versus C1PC for read-heavy load in Figure VI.5(a), the maximum throughput more than doubles while satisfying the same response time demands. For the write-heavy load in Figure VI.5(b), the maximum throughput is increased by 60 – 80% for both time limits.

## VI.7 Analysis

To verify the results from the simulations, the analytical model in [14] is used. The model consists of a multiple-class, open queuing network. The inter-arrival times are exponentially distributed. Single-server queuing theory can be used by assuming uniformity for all nodes [12]. The steady-state solutions are derived using results from  $M/G/1$  queues [12].

Figure VI.6 shows a comparison of simulation and analysis results for the SUSR protocol. The analyzes are shown in black and the simulations in white. The read-intensive results are plotted as diamonds, while the write-intensive are plotted as squares. The figure shows that the shape of the plots seem to verify the simulations. The largest difference for response time is less than 10%, which strongly indicates that the results are accurate.

## VI.8 Conclusion and Further Work

In this paper, approaches for efficient execution of simple read-only and write-only transactions are presented. The context of the paper is main-memory databases where replication ensures durability and high availability. Both simulations and statistical analyzes were performed. The results show that, when applicable, the SUSR protocols significantly reduce the delay and increase the throughput. Compared to C1PC, SUSR tolerates 150% more transactions during read-heavy load while keeping the response time below 5 ms for 95% of the transactions. For write heavy load, the improvement is around 70%. The results indicate that the protocols can be favorably applied in systems which require high availability and short real-time responses and a large part of the load consists of transactions which only read or write a single record, such as telecommunications, sensor networks and online games.

Further work includes implementing SUSR in a real-world system and adopting the approach to multiple backups.

## References

- [1] D. Agrawal and V. Krishnaswamy. Using multiversion data for non-interfering execution of write-only transactions. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 98–107, New York, NY, USA, 1991. ACM.
- [2] P. A. Bernstein, J. B. Rothnie, N. Goodman, and C. A. Papadimitriou. The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case). *IEEE Transactions on Software Engineering*, 4(3):154–168, 1978.
- [3] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publ. Co., Inc., 1987.
- [4] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, 11(2):205–212, 1985.
- [5] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 04(6):509–516, 1992.
- [6] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions

- in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, 1982.
- [7] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [8] Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972, pages 87–100, Le Mont-Saint-Michel, France, 1995. Springer-Verlag.
- [9] Abdelsalam Abdelhamid Heddaya. *Managing event-based replication for abstract data types in distributed systems*. PhD thesis, Harvard University, Cambridge, MA, USA, 1988.
- [10] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, 1995.
- [11] ISO. Information Technology - Database Language SQL - part 4: Persistent Stored Modules (SQL/PSM). ISO/IEC 9075-4, 2003.
- [12] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley & sons, 1991.
- [13] Heine Kolltveit and Svein-Olaf Hvasshovd. The Circular Two-Phase Commit Protocol. In *Proceedings of International Conference of Database Systems for Advanced Applications*, pages 249–261. Springer-Verlag, 2007.
- [14] Heine Kolltveit and Svein-Olaf Hvasshovd. Performance of Main Memory Commit Protocols. Technical Report 06/2007, NTNU, IDI, 2007.
- [15] Heine Kolltveit and Svein-Olaf Hvasshovd. Efficient High Availability Commit Processing. *ARES*, 2008.
- [16] Heine Kolltveit and Svein-Olaf Hvasshovd. Main memory commit processing: The impact of priorities. *Proceedings of the International Conference of Database Systems for Advanced Applications*, 2008.
- [17] Jan Lindström and Tiina Niklander. Benchmark for real-time database systems for telecommunications. In *DBTel '01: Proceedings of the VLDB*

- 2001 International Workshop on Databases in Telecommunications II*, pages 88–101, London, UK, 2001. Springer-Verlag.
- [18] Jan Lindström, Tiina Niklander, Pasi Porkka, and Kimmo E. E. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Proceedings of the International Workshop on Databases in Telecommunications*, pages 158–173, London, UK, 2000. Springer-Verlag.
- [19] Jørgen Løland and Svein-Olaf Hvasshovd. Online, non-blocking relational schema changes. In *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 405–422. Springer Berlin, 2006.
- [20] Baojing Lu, Qinghua Zou, and William Perrizo. A dual copy method for transaction separation with multiversion control for read-only transactions. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 290–294, New York, NY, USA, 2001. ACM.
- [21] MobileTracker. <http://www.mobiletracker.net/archives/2005/05/18/mobile-subscribers-worldwide>, 2005.
- [22] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R\* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.
- [23] T. Niklander, J. Kiviniemi, and K Raatikainen. A real-time database for future telecommunication services. In D Gaïti, editor, *Intelligent Networks and Intelligence in Networks*. Chapman & Hall, 1997.
- [24] Bernd Page and Wolfgang Kreutzer. *The Java Simulation Handbook. Simulating Discrete Event Systems with UML and Java*. Shaker Verlag, 2005.
- [25] Krithi Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.
- [26] Reuters. <http://investing.reuters.co.uk/news/articleinvesting.aspx?type=media&storyID=nL29172095>, 2007.
- [27] Mikael Ronström. *Design and Modelling of a Parallel Data Server for Telecom Applications*. PhD thesis, Uppsala University, 1998.

- 
- [28] Mikael Ronström. Database requirement analysis for a third generation mobile telecom system. In *Proceedings of the International Workshop on Databases in Telecommunications*, pages 90–105, London, UK, 2000. Springer-Verlag.