City Research Online

City, University of London Institutional Repository

Citation: Stankovic, V. (2008). Performance Implications of Using Diverse Redundancy for Database Replication. (Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/17440/

Link to published version:

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

City Research Online: http://openaccess.city.ac.uk/ publications@city.ac.uk

Performance Implications of Using Diverse Redundancy for Database Replication

by

Vladimir Stankovic

v.stankovic@city.ac.uk

Centre for Software Reliability

City University

London EC1V 0HB

United Kingdom

February 2008

List of Contents

1. Introduction 1

1.1. Motivations and Aims 1

1.2. Summary of Work 3

1.3. Thesis Outline 5

2. Concepts and Background 6

2.1. Fault Tolerance via Diverse Redundancy 6

2.2. Database Definitions 9

2.2.1. Transactions 9

2.2.2. Isolation Levels 11

2.2.3. Concurrency Control and Correctness Criteria 13

2.2.4. Liveness 16

2.3. Database Replication 17

2.3.1. ROWAA-Based Replication 18

2.3.2. Correctness in Replicated Databases 19

2.3.3. Conflicts and Deadlocks 23

2.3.4. Transaction Atomicity 24

2.4. TPC-C – an On-Line Transaction Processing Benchmark 25

3. Architecture of DivRep Middleware 28

3.1. DivRep – Replication with Diverse Database Servers 28

3.1.1. Dependable Replication Algorithm (DRA) 32

3.1.2. DRA Optimisations 40

3.1.3. Distributed Deadlock Avoidance 42

3.2. Correctness of DRA 46

3.2.1. Safety 46

3.2.2. Liveness 48

3.3. Hybrid Approach of DivRep 49

3.4. Discussion 50

3.4.1. Comparing DivRep to Other Replication Techniques 51

3.4.2. Possible Changes to DivRep 53

4. Experimental Evaluation of DivRep Performance 56

4.1. Test Harness 57

4.2. Preliminary Experiments – Systematic Differences in the Performance of Diverse Servers ... 64
4.3. When Diverse Redundancy Performs Better than Non-Diverse Redundancy . 68
4.3.1. Confidence in the Results ... 70
4.3.2. Performance Comparison of Different DBMS Configurations ... 70
4.4. Performance Implications of Improving Dependability ... 74
4.4.1. SI-Rep Simulation ... 75
4.4.2. DivRep vs. a ROWA-Based Replication (SimSI-Rep) ... 76
4.4.3. Discussion of DivRep vs. SimSI-Rep Comparison ... 84
4.4.4. User-Centric Analysis ... 86
4.5. Minimising Replication Overhead Using Priority Mechanisms ... 100
4.5.1. The Problem ... 100
4.5.2. The Solution ... 103
4.5.3. Discussion ... 107
4.5.4. Related Work ... 110
5. Uncertainty-Explicit Assessment of DivRep Components ... 111
5.1. Motivation for Using Uncertainty-Explicit Assessment ... 112
5.2. Bayesian Approach to Assessment of a Single Attribute ... 114
5.3. A Model for Assessment of 2 Non-Independent Attributes ... 115
5.4. A Numerical Example ... 117
5.4.1. Prior Distributions ... 119
5.4.2. Observations ... 120
5.4.3. Posteriors ... 121
5.5. Discussion and Related Work ... 123
6. Related Work ... 126
6.1. A Multitude of Database Replication Solutions ... 126
6.2. Load Balancing and Adaptability ... 130
6.3. Consistency Guarantees ... 133
7. Conclusions ... 135
7.1. Research Assessment ... 136
7.2. Future Directions ... 138
Bibliography ... 141
List of Acronyms ... 152
Appendix A ... 153

Database Schema of the Log Database 153

List of Tables

TABLE 4-1 80
TABLE 4-2 81
TABLE 4-3 91
TABLE 4-4 91
TABLE 4-5 99
TABLE 4-6 107
TABLE 5-1 116
TABLE 5-2 120
TABLE 5-3 121
TABLE 5-4 122

List of Figures

FIGURE 2-1 ... 24
FIGURE 3-1 ... 29
FIGURE 3-2 ... 30
FIGURE 3-3 ... 31
FIGURE 3-4 ... 32
FIGURE 3-5 ... 34
FIGURE 3-6 ... 41
FIGURE 3-7 ... 42
FIGURE 3-8 ... 43
FIGURE 3-9 ... 45
FIGURE 3-10 ... 46
FIGURE 4-1 ... 58
FIGURE 4-2 ... 66
FIGURE 4-3 ... 67
FIGURE 4-4 ... 68
FIGURE 4-5 ... 71
FIGURE 4-6 ... 71
FIGURE 4-7 ... 73
FIGURE 4-8 ... 73
FIGURE 4-9 ... 74
FIGURE 4-10 ... 79
FIGURE 4-11 ... 79
FIGURE 4-12 ... 88
FIGURE 4-13 ... 96
FIGURE 4-14 ... 96
FIGURE 4-15 ... 97
FIGURE 4-16 ... 97
FIGURE 4-17 ... 98
FIGURE 4-18 ... 98
FIGURE 4-19 ... 102
FIGURE 4-20 ... 103
FIGURE 4-21 ... 106
FIGURE 5-1 ... 114

To my parents, for their love.

Acknowledgements

I would like to express my gratitude to several people who have, in distinct ways, contributed to producing the research work.

I would like to thank my supervisor, Peter Popov, for the indispensable help in many aspects. First, I thank him for offering me the opportunity to conduct the research. He provided continuous support and conveyed great enthusiasm throughout the course of the research. Together with his prompt and precise guidance, the motivation helped me in outstanding the obstacles of the research work.

I wish to thank Ilir Gashi for his friendship and thoughtful remarks on many facets of research. Sharing the working space with him made the research life easier.

While presenting the work at local departmental meetings, my colleagues in CSR offered scientific advice and suggested constructive alternatives for the research. This helped me gain a deeper insight and improve the research work. Optimistic comments from Andrey Povyakalo encouraged me in persevering during the writing up. Peter Bishop provided useful comments on the presentation of the thesis. I am grateful to Basi Isaacs for the smooth running of the administrative parts of the research life.

I am thankful to Prof. Ricardo Jimenez-Peris and Prof. Alexander Romanovsky for accepting the invitation to be the external examiners.

Special thanks goes to my dear friend Marko Ivin, whom I am indebt for fruitful technical discussions and, more importantly, the encouragement throughout the research.

In addition, I would like to thank Phil Parkin for useful editorial suggestions for the parts of the thesis.

Last, but certainly not least, I am particularly grateful to my family and my wife for their love. They have been supportive and understanding; the effort would not have been possible without them.

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Abstract

Using *diverse redundancy* for *database replication* is the focus of this thesis. Traditionally, database replication solutions have been built on the fail-stop failure assumption, i.e. that crashes are believed to cause a majority of failures. However, recent findings refuted this common assumption, showing that many of the faults cause systematic non-crash failures. These findings demonstrate that the existing, non-diverse database replication solutions, which use the same database server products, are ineffective fault-tolerant mechanisms. At the same time, the findings motivated the use of diverse redundancy (when different database server products are used) as a promising way of improving *dependability*. It seems that using a fault-tolerant server, built with diverse database servers, would deliver improvements in availability and failure rates compared with the individual database servers or their replicated, non-diverse configurations.

Besides the potential for improving dependability, one would like to evaluate the performance implications of using diverse redundancy in the context of database replication. This is the focal point of the research. The work performed to that end can be summarised as follows:

- We conducted a substantial performance evaluation of database replication using diverse redundancy. We compared its performance to the ones of various non-diverse configurations as well as non-replicated databases. The experiments revealed systematic differences in behaviour of diverse servers. They point to the potential for performance improvement when diverse servers are used. Under particular workloads diverse servers performed better than both non-diverse and non-replicated configurations.
- We devised a middleware-based database replication protocol, which provides dependability assurance and guarantees database consistency. It uses an *eager update everywhere* approach for replica control. Although we focus on the use of diverse database servers, the protocol can be used with the database servers from the same vendor too. We provide the correctness criteria of the protocol. Different regimes of operation of the protocol are defined, which would allow it to be dynamically optimised for either dependability or performance improvements. Additionally, it can be used in conjunction with high-performance replication solutions.
- We developed an experimental test harness for performance evaluation of different database replication solutions. It enabled us to evaluate the performance of the diverse database replication protocol, e.g. by comparing it against known replication solutions. We show that, as expected, the improved dependability exhibited by our replication protocol carries a performance overhead. Nevertheless, when optimised for performance improvement our protocol shows good performance.
- In order to minimise the performance penalty introduced by the replication we propose a scheme whereby the database server processes are prioritised to deliver performance improvements in cases of low to modest resource utilisation by the database servers.
- We performed an *uncertainty-explicit* assessment of database server products. Using an integrated approach, where both performance and reliability are considered, we rank different database server products to aid selection of the components for the fault-tolerant server built out of diverse databases.

1. Introduction

There never were, in the world, two opinions alike, no
more than two hairs, or two grains;
the most universal quality is diversity.
Michel Eyquem de Montaigne

1.1. Motivations and Aims

A vast family of computer technologies, commonly referred to as Commercial-Off-The-Shelf (COTS) products, has emerged as the alternative to building bespoke developments. The use of COTS is stimulated by a desire to reduce overall system development cost and time to deployment. Although using COTS software has become pervasive in the past two decades and many governments and businesses mandate their use, doubts are often raised due to the increased cost of integration and insufficient level of dependability. The latter has motivated the research on using *software fault tolerance*, as the only viable way of obtaining the required system dependability when COTS components are used (Popov, Strigini et al. 2000), (Hiltunen, Schlichting et al. 2000), (Valdes, Almgren et al. 2003).

The field of fault tolerance is well-established (Anderson and Lee 1990). This is true for hardware fault tolerance as well as for its software counterpart (Lyu 1995): both have been providing techniques for building highly reliable, continuously available and extremely safe software. There are many proven concepts of software fault tolerance, such as recovery blocks (Randell 1975), used primarily for error detection (beside facilitating recovery and providing continuity of service), but also more mature and able forms: *N*-version-programming, of which a replication with diverse products is an instance (Avizienis and Kelly 1984) or self-checking modular redundancy (Laprie, Béounes et al. 1990). Using N-version programming in the form of diverse modular redundancy became more practicable with the advent of COTS components.

Note that despite the possibility of classifying Off-The-Shelf (OTS) components as commercial or non-commercial (Popov, Strigini et al. 2000), the cost incurred by

either is significantly less than if a corresponding purpose-built product was being developed. The distinction between the two families of components is, therefore, unimportant for the research described in this thesis. In addition, despite the term COTS being commonly used to refer to both software and hardware products, the work in this thesis focuses on the former.

Using products diverse by design in a fault-tolerant configuration (*diverse redundancy* or *design diversity*) with the aim of increasing system dependability is an attractive idea. Diverse redundancy can be applied to a number of (complex) software e.g. application or web servers, Business Process Execution Language (BPEL) engines etc. The effectiveness of design diversity will depend on the achievable dependability gains, implementation difficulties and associated cost (developmental, operational etc.). Recently, a thorough research (Gashi, Popov et al. 2007), (Gashi, Bishop et al. 2007) was conducted to explore and estimate the possible advantages of using diverse redundancy with Database Management Systems (DBMS), a category of complex COTS. The results obtained showed that considerable dependability gains can be achieved. Some of the most distinct findings are as follows:

- Using the fault (“bug”) scripts from four DBMSs and subsequently executing them against each, Gashi et al. demonstrated that very few bugs cause failures in more than one product and none causes failures in more than two. Hence, a simple two-diverse configuration using different DBMS products would *detect* most of the failures.
- If a single DBMS product is replaced with a 1-out-of-2 diverse redundant system, an order of magnitude increase in reliability gains is possible.

These results warrant the use of diverse redundancy for improved dependability when a particular family of products is considered, i.e. DBMSs. Dependability gains aside, the use of redundancy has inevitable implications on system performance (Gray, Helland et al. 1996). The same applies to diverse redundancy as a particular form of redundancy. The research described in this thesis focuses specifically on investigating the performance implications of using diverse redundancy in the context of database replication. The underlying principle of database replication is database consistency, which guarantees that users perceive a replicated system as a reliable and available centralised database with adequate performance. This requirement poses difficulties that are exacerbated when diverse servers are used. Therefore, this thesis aims to help

database practitioners in answering the following questions with respect to *practicality* and *performance* of database replication with diverse DBMSs:

- *Performance*:
  - What is the magnitude of performance penalty incurred when database replication is based on diverse redundancy?
  - Are there any characteristics of diverse database servers that can be exploited for performance gain? Can these characteristics bring performance benefits under (at least) relaxed dependability requirements?
  - Are there ways of trading off dependability assurance for performance improvement?
- *Practicality*:
  - How can data consistency be ensured when diverse database servers are used for database replication?

1.2. Summary of Work

The work described in the thesis has been performed as a part of the wider research initiative in the Centre for Software Reliability (City University London) to evaluate the usefulness of diverse redundancy for potential improvements in dependability and performance of COTS software components. In this respect, the central point of this thesis is evaluation of performance implications of such an approach applied to database replication. In particular, the research work encompasses the following:

A replication algorithm that ensures database consistency. We have devised a novel replication algorithm that ensures eager, update-everywhere replication and guarantees consistency of diverse databases. Little complexity is added to the concurrency control mechanisms of underlying databases to achieve this. In this way no significant performance overhead due to possibly complex replication protocol is incurred. A proof of correctness of the algorithm has been provided.

Middleware-based database replication with diverse servers. A practical middleware-based database replication solution that uses diverse redundancy has been developed. Two main regimes of operation of the replication solution are identified, tailored for either improving dependability or performance. Also, a hybrid approach is proposed, in which the two regimes of operation are brought together to provide integrated quality of service. The replication solution advocates that a pair of diverse database

servers be employed in a fault-tolerant configuration (FT-node). Certain optimisation techniques have been introduced in the middleware to improve performance of the FT-node.

Extensive performance evaluation of the proposed replication solution. The research described in the thesis includes extensive experimental results of performance implications when diverse database servers are used for replication. We obtain our experimental results on a *real* implementation that evaluates the feasibility of the middleware-based solution in the *real environment*. Despite inherent complexity this experimental method is preferred over simulation studies that have limited capabilities and often produce misleading results (Jain 1991). The experimental evaluation includes a comparison of diverse and non-diverse database servers when the replication solution is optimised for either maximum dependability or maximum performance. The main findings of the experimental studies are as follows:

- The results of preliminary experiments (Gashi, Popov et al. 2004) demonstrate that systematic differences between performances of diverse servers exist, i.e. one of the servers exhibits better performance on particular (sets of) SQL operations. This observation suggests that performance improvement might be observed when a pair of diverse database servers is used: if the execution of SQL operations is distributed in the way that each server executes only the portions of operations it is faster on, the diverse pair performance will be better than the performance of individual servers.
- Under a workload with a single modifying client, a significant performance gain can be obtained with a pair of diverse database servers when compared to non-diverse pairs or single server configurations (Stankovic and Popov 2006).
- We compare our replication solution against an instance of Read-Once-Write-All-Available (ROWAA) (Bernstein, Hadzilacos et al. 1987), a well-known replication approach. The results show that under specific application conditions, dependability assurance might be costly due to the performance penalty it introduces. We propose a solution for decreasing the performance penalty. Also, we identify significant difference in the performance of diverse servers as an important factor in the recorded performance deterioration.

Uncertainty-explicit assessment for selection of DivRep components. The probabilistic model using Bayesian inference (Littlewood, Popov et al. 2000) was adapted and applied in assessing a single software product from two perspectives:

- performance, represented by *timeliness* of the product's results and
- reliability, represented by *correctness* of the product's results.

The model is applied for ranking a set of DBMS products using experimental data (based on the TPC-C benchmark (TPC 2002a)) as evidence. In this way the selection of the components to be included in an FT-node has been conducted using *both* reliability and performance.

1.3. Thesis Outline

The thesis is organized as follows: Chapter 2 gives background on design diversity and database replication and it details definitions and explains terms regarding DBMSs. It introduces an industry standard benchmark from the Transaction Processing Council, TPC-C (TPC 2002a), used as the basis for the experimental evaluation performed in the course of the research. Chapter 3 describes the architecture of the middleware-based replication solution that uses diverse database servers as underlying components. The replication algorithm that ensures database consistency is explained in this section and a proof of its correctness is provided. Detailed discussions on the comparison with the existing replication protocols, as well as possible optimisations of the replication solution are given in this chapter. Chapter 4 reports the extensive experimental evaluation of diverse servers' performance and the proposed replication solution. We show a potential for performance improvement through use of diverse servers revealing systematic differences in server response time. Under certain types of workload a pair of diverse servers performs better than non-redundant or non-diverse server configurations. We compare the performance of the replication protocol to a solution based on ROWA replication. Also, the results of minimising the performance overhead introduced by the replication are presented. In Chapter 5 we present a Bayesian model for assessing attributes of COTS components. We use the assessment method in an empirical study to rank database servers in order to select the ones to be included in FT-node. Chapter 6 contains a critical review of the related literature. The emphasis is on database replication. Finally, Chapter 7 summarises the main conclusions and suggests directions for future work.

2. Concepts and Background

Physical concepts are free creations of the human mind, and are not, however it may seem, uniquely determined by the external world.

Albert Einstein

2.1. Fault Tolerance via Diverse Redundancy

Dependability (Laprie, Randell et al. 2004) is the term used to precisely describe those system properties that allow us to rely on a system functioning as required. Dependability includes, among other attributes, reliability, safety, security, and availability. From the dependability point of view, we will be concerned in particular with the reliability of the database replication solutions.

When systems are built using "off-the-shelf" products, fault tolerance is often the only viable way for obtaining required system dependability (Popov, Strigini et al. 2000), (Hiltunen, Schlichting et al. 2000). Fault-tolerant systems are able to continue operation properly in the event of failures, after some faults have manifested themselves. Hence the concept of *fault tolerance* (Anderson and Lee 1990) assumes that faults are present in the system, and that it is possible for the system to handle them without external interventions. The goal of fault tolerance is to ensure that system faults do not result in system failure. There is, however, an inherent cost in building and maintaining fault tolerant systems, e.g. due to developing multiple versions of complex software. This cost is usually high and therefore it tends to be employed mostly in applications where a system failure would cause catastrophic accidents, perhaps resulting in a loss of life, or where a system failure would lead to large economic losses. Fault tolerance can be achieved through both software and hardware. One of the best known hardware fault-tolerance techniques is Triple Modular Redundancy (TMR). TMR builds on the early work by the computer pioneer Johann Von Neumann who advocated the use of redundancy, in the form of Triple Redundancy (von Neumann 1956), as a fault tolerant technique. A simplified description of Triple Redundancy is as follows: three systems perform a process and the results are processed by a voting system (*voter*) to produce a single output. A

drawback of Triple Redundancy is that a failure of the voter causes the overall system failure. To remedy the single point of failure, TMR configuration employs three identical voting systems instead of one. Note that the use of term *voter* is somewhat imprecise – it is the underlying redundant systems that perform the voting rather than the voting system itself, which instead makes a decision based on the votes. Thus, possibly, a more accurate term for a voting system would be a *decider* (N.B. von Neumann called it a *majority organ*). The concept of TMR has been also applied to software redundancy; the most notable example is *N*-Version Programming (NVP) (Avizienis and Chen 1978). NVP uses redundancy, i.e. multiple functionally equivalent programs are independently created from the same initial specification, with the aim to improve reliability of a system. The assumption is that when one of the channels fails the others will perform correctly.

We are concerned with the use of *diverse* database servers for building database replication solution that exhibits improved dependability. *Diverse redundancy*, often referred to as *design diversity*, involves *bespoke development* or *reuse* of multiple diverse versions of a piece of software with the goal of increasing availability or reliability of a system. The saying "Two heads are better than one" expresses the widespread belief that the use of redundancy and diversity is a suitable way for reducing the risk of failures. Charles Babbage (Babbage 1974) advocated such a principle by stating that humans would more likely trust the results of complex arithmetic calculation if two persons have arrived independently at the same output.

The premise that software suffers exclusively from *design faults* and not from *physical faults*, which are hardware specific, has been known for many years. Let us consider the simplest definition of a design fault to be the following: "a fault that is introduced in software during its development". This implies that design faults will be simply replicated if non-diverse redundant copies of the same software product are used. Such a fault-tolerant mechanism is incapable of protecting the system against design faults. The ideal claim, on the other hand, of employing diverse redundancy would be: "Software product A does not fail when software product B does". The *lack of dependence* between the failure modes of the software products would be highly desirable – if it existed, one could claim that the probability of failure of the diverse system can be calculated by just conservatively multiplying the probability of failures of the individual software products in the diverse system. However virtually all of the experimental studies conducted for measuring benefits of diverse redundancy, such as

(Knight and Leveson 1986), (Kelly and Avizienis 1983) and (Eckhardt, Caglayan et al. 1991), demonstrated that this goal is unlikely to be attainable in practice. Despite the experiments pointing to the lack of independence of failures between different software versions, there is evidence that diverse redundancy would deliver some increase in reliability compared to using a software system built out of a single version. The review in (Littlewood, Popov et al. 2001) explores the extent of such an increase with a particular focus on the modelling of reliability of systems using diverse redundancy.

Gashi et al. (Gashi, Popov et al. 2007) have experimentally evaluated the potential for dependability gains from diverse redundancy when using DBMSs. The results suggest that diverse redundancy would be effective for tolerating design faults. By experimenting with publicly available fault reports of diverse DBMSs, the authors showed that a very small percentage of the collected faults would cause coincident failures in diverse database servers - only in very few cases, a demand that triggers a bug in one server would cause failure in another one. Also, no demand caused a coincident failure in more than two servers. Therefore, a fault-tolerant server built with two diverse servers is likely to ensure a high *failure detection* rate.

Further means of achieving effective fault tolerance are presented in (Gashi and Popov 2006). The authors propose a *data diversity* approach, based on the work of (Ammann and Knight 1988). Data diversity in DBMSs is possible due to the redundancy in the SQL language. The underlying idea is that one, or several, SQL operations can be "rephrased" into a workaround, a syntactically different but semantically equivalent sequence, to produce redundant executions. By executing the workarounds, the failures are less likely, e.g. the rephrased operation might follow different execution path from the original one and the failure would be avoided. The authors defined a set of rephrasing rules that would tolerate at least 60% of the faults examined in the study. In addition to failure detection, they showed that the rephrasing can aid failure diagnosis (identification of the failed DBMS) and recovery of the state of the failed product.

2.2. Database Definitions

2.2.1. Transactions

We are concerned with database transactions, logical units of work within a database management system that are treated reliably and independently of each other. A transaction represents a unit of interaction with a DBMS and consists of any number of *read* and *write* operations and finishes with either *commit* or *abort*. Let D = $\{x_1, x_2, \ldots, x_n\}$ be a representation of data items stored in a database and let $r(x_k)$ and $w(x_k)$ be a read and a write operation on data item x_k: $x_k \in \text{D}$ respectively, and let c and a be the commit and abort operations. We define (Bernstein, Hadzilacos et al. 1987) a transaction T_i to be a partial order with ordering relation $<_\text{i}$ where:

1. $T_i \subseteq \{r_i(x_k), w_i(x_k) | x \in \text{D}\} \cup \{a_i, c_i\}$;
2. $a_i \in \text{T}_\text{i}$ **iff** $c_i \notin \text{T}_\text{i}$;
3. let o be a_i or c_i, whichever is in T_i, for all other operations $o' \in T_i : o' <_\text{i} o$; and
4. if $r_i(x_k), w_i(x_k) \in T_i$ then either $r_i(x_k) <_\text{i} w_i(x_k)$ or $w_i(x_k) <_\text{i} r_i(x_k)$;

An implicit assumption is made in the above model: a transaction writes a particular data item only once. This is the reason why in the property *4.* a pair of write operations is not considered.

We use a structure called *history* (Bernstein, Hadzilacos et al. 1987) to model (concurrent) execution of transactions. It indicates the relative order in which the operations of transactions have executed. Since the operations might execute concurrently, a *history* is defined as a partial order. Let $T = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions. A *complete history H* over T is a partial order with ordering relation $<_\text{H}$ where:

- $H = \bigcup_{i=1}^{n} T_i$;
- $\bigcup_{i=1}^{n} <_\text{i} \subseteq <_H$; and
- for any two conflicting operations $p, q \in H$, either $p <_H q$ or $q <_H p$;

Following the established criterion (Papadimitriou 1986), a conflict between two transactions is defined as follows: if two operations, belonging to different concurrently executing transactions, read or write the same data item and not both are read operations, the corresponding transactions conflict.

A *history* is a prefix of a *complete history*. While a *history* could contain a transaction that has neither committed nor aborted (*active* transaction), a *complete history* contains no such transactions. We are specifically interested in the *committed projection* of history *H*, denoted as *C(H)*: it is the history obtained from *H* by removing all operations that do not belong to any of the committed transactions in *H*. *C(H)* is a complete history over the set of committed transactions in *H*. We provide a definition of a partial order prefix with the aim to disambiguate the term (Bernstein, Hadzilacos et al. 1987):

Prefix P` = $\{\sum`, <`\}$ is a prefix of a partial order P = $\{\sum, <\}$, where $\sum$ is the domain of the partial order P and < is an irreflexive, transitive binary relation on $\sum$, if:

- $\sum` \subseteq \sum$
- $\forall e_i \in \sum`$, $e_1 <` e_2$ **iff** $e_1 < e_2$
- $\forall e_i \in \sum`$, if $\exists e_j \in \sum$ and $e_j < e_i$, then $e_j \in \sum`$

A first formal discussion of database transaction properties can be found in (Gray 1981). Since then a standard approach has emerged in the literature through ACID properties. The acronym ACID stands for the following:

- *Atomicity* – ability to guarantee that either all of the tasks of a transaction are performed or none of them is.
- *Consistency* – ability to preserve the legal states imposed by the *integrity constraints*. More informally, this means that no rules are broken as a consequence of transaction execution.
- *Isolation* - ability to make operations in a transaction appear isolated from all other operations executed by other transactions. This property guarantees that, although transactions could execute concurrently, the outcome of the execution is equal to the outcome of a serial transaction execution.
- *Durability* – ability to guarantee that changes made by a transaction are permanent once the transaction successfully completes (*commits*).

We are interested in *distributed transactions*, which are characterised by the execution of one or more operations that, individually or as a group, update data on two or more

distinct nodes of a distributed database. They are commonly observed in replicated database systems. A distributed transaction must provide ACID properties among multiple participating databases, which are dispersed among different physical locations. The isolation property poses a special challenge for multi database transactions, since the requirement that transactions execute in a serial manner is exacerbated in a distributed setting.

2.2.2. Isolation Levels

Of particular interest to our work is the *isolation* property (it has appeared for the first time under the term Degrees of Consistency in (Gray, Lorie et al. 1975)). Isolation guarantees that if a conflict between transactions is possible then the transactions must be isolated from each other. Different types of isolation have been proposed. The ANSI SQL standard specifies four levels of isolation (ANSI 1992): s*erializable*, *repeatable read*, *read committed* and *read uncommitted*. The highest level of isolation is the *serializable* level, which requires every history to be equivalent to a *serial* history, i.e. history in which transactions appear to have executed one after another without overlapping. Lower isolation levels are less restrictive but they can introduce inconsistencies during transaction executions, i.e. they offer better performance at the expense of compromising consistency. Due to its impact on system performance, isolation is the most frequently relaxed ACID property.

In order to characterise transactional isolation property and allow for different implementations ANSI SQL defines three *phenomena*. A description of the phenomena is given in (Berenson, Bernstein et al. 1995) and an additional one, *dirty writes*, is specified:

- *Dirty writes* – Assume a transaction, T_i, modifies a data item, and another transaction, T_j, then modifies the same data item before T_i ends (commits or aborts). Subsequently, after T_j ends, it will be unclear what the correct value of the data item should be. Subsequently, if any of the two transactions perform an abort, it is unclear what the correct data value should be.
- *Dirty reads* – Assume a transaction, T_i, modifies a data item, and another transaction, T_j, then reads the data item before T_i performs a *commit* or *abort*. If T_i then performs an *abort*, T_j has read a data item that never really existed because it was never committed.

- *Non-repeatable (fuzzy) reads* – Assume a transaction T_i reads a data item and another transaction, T_j, then modifies or deletes that data item and *commits*. If T_i then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.
- *Phantoms* – Assume a transaction T_i reads a set of data items satisfying some *search condition*. Transaction T_j then creates data items that satisfy T_i's *search condition* and commits. If T_i then repeats its read with the same *search condition,* it gets a set of data items different from the first read.

Preventing *dirty writes* is a prerequisite for database consistency and automatic transaction rollback (Gray, Lorie et al. 1975), (Berenson, Bernstein et al. 1995). The ANSI isolation levels are defined in terms of the above phenomena, and in particular, according to the ones they are disallowed to experience:

- *Read uncommitted* prevents *dirty write*.
- *Read committed* prevents *dirty writes* and *dirty reads*.
- *Repeatable read* prevents *dirty writes*, *dirty reads* and *non-repeatable reads*.
- *Serializable* prevents *dirty writes*, *dirty reads*, *non-repeatable reads* and *phantoms*.

The ANSI isolation levels have been criticised in (Berenson, Bernstein et al. 1995) and (Adya, Liskov et al. 2000) because they do not accurately capture the isolation levels offered by many database management systems. The work has shown that the three phenomena defined by ANSI are ambiguous and they fail to characterise all possible anomalous behaviour of different isolation levels. The work in (Berenson, Bernstein et al. 1995) defines an additional isolation level, *snapshot isolation (SI)*, which is offered in leading commercial and open-source database systems (Oracle, Microsoft SQL Server, with certain variations, PostgreSQL etc.). Snapshot isolation avoids the phenomena defined in ANSI but exhibits inconsistent behaviour in some situations because it can produce other types of anomalies, such as *write skew* (Berenson, Bernstein et al. 1995) and *read skew* (Fekete, Liarokapis et al. 2005), (Fekete, O'Neil et al. 2004). The two anomalies can be described as follows:

- *Write skew* - Assume two data items, x and y, are related by a constraint that $x + y > 0$, and the initial values of the two data items satisfy the constraint. Further assume that the following order of operations, belonging to two transactions, T_i and T_j, is executed: $r_i(x)$, $r_i(y)$, $r_j(x)$, $r_j(y)$, $w_i(x)$, $w_j(y)$. It is possible that the

transactions modify the data items in the way that the constraint is violated, e.g. T_i sets *x* to 100, but T_j sets *y* to -150.

- *Read skew* - Assume two data items, *x* and *y*, are related by a constraint that $x + y > 0$, and the initial values of the two data items satisfy the constraint. Further assume that the following order of operations, belonging to two transactions, T_i and T_j, is executed: $r_i(x), w_j(x), w_j(y), c_j, r_i(y)$. It is possible that the reading of the sum x + y by transaction T_i returns a result that violates the constraint.

2.2.3. Concurrency Control and Correctness Criteria

Concurrency control mechanisms in DBMSs ensure that transactions execute concurrently without violating data integrity of a database. The main goal of concurrency control mechanisms is providing different degrees of isolation to transaction execution. However they should also prevent concurrent executions, which exhibit worse performance than a serial execution (Second Law of Concurrency Control (Gray and Reuter 1993)).

A component in a DBMS, referenced as a *scheduler*, manages the overlapping executions of transactions. A scheduler receives operations from users and makes sure that they are executed in a *correct* way, according to the specified isolation levels. Typically, *correctness* implies that an execution of a set of concurrent transactions produces a serializable history, i.e. one that demonstrates the same effects as a serial execution of the same set of transactions.

Serializability theory (Bernstein, Hadzilacos et al. 1987) has been developed to provide criteria for deciding whether a history is serializable. The concept of *equivalence* was introduced in order to provide syntactical rules for transforming one history to another. In this way it becomes possible to determine if two histories have the same effect and if a history is serializable. In database concurrency control literature there are in fact two established approaches for deciding equivalence of histories and deciding if a history is serializable: *conflict serializability* and *view serializability* (Papadimitriou 1986), (Bernstein, Hadzilacos et al. 1987). The former is usually applied to concurrency control of a single-version DBMSs, while the latter is used for multi-version DBMSs.

In conflict serializability two operations are considered to conflict if they both access the same data item and at least one of them is a write. The end result will depend on the order of execution of two conflicting operations. It should be noted that the same

result would be produced if the order of non-conflicting operations, belonging to different concurrent transactions, was interchanged in the history. The order imposed by conflicting operations determines dependencies of precedence between the transactions that contain those operations. A structure called *serialization graph* captures these dependencies for a history *H*. It is a directed graph denoted as *SG(H)*. The graph contains one node for each transaction in the committed projection of the history; there exists an edge between T_i and T_j if and only if there are two conflicting operations, p_i and q_j such that p_i comes before q_j. It can be shown that two histories are conflict-equivalent if their serialization graphs are identical. A history is serializable if the serialization graph has no cycles.

On the other hand, two histories, *H* and *H'*, are view-equivalent if:

- They are over the same set of transactions and have the same operations.
- They have the same *reads-from* relation: for any two committed transactions T_i and T_j and for any data item *x*, if T_i reads *x* from T_j in *H* then T_i reads *x* from T_j in *H'*.
- They have the same *final writes*: for each data item *x*, if w_i is the final write of *x* in *H* then it holds for *H'* too.

Then view-serializability is defined as follows: A history *H* is view-serializable if for any *prefix H'* of *H*, the committed projection *C(H')* is view equivalent to some serial history.

Traditionally, *locking-based* protocols (Bernstein, Hadzilacos et al. 1987), (Gray and Reuter 1993) have been used to implement different isolation levels. In particular, strict 2-Phase locking (S2PL) has been traditionally used to implement the serializable isolation level, though the ANSI SQL standard (ANSI 1992) does not mandate its use – another type of mechanism can be used to provide serializable isolation level as long as in an interleaved execution the transactions see the same values and leave the same final state as is the case in a serial execution. S2PL avoids the phenomena described in the ANSI standard (ANSI 1992) and also the ones identified in (Berenson, Bernstein et al. 1995). Both read and write locks are acquired: a *shared lock* when reading a data item and an *exclusive lock* when writing a data item. The former, acquired by a transaction on a data item, allows only the access of concurrent reads, while the latter prevents both reading and writing of the same data item by other transactions. All locks are released at the end of the transaction, following the commit or abort operation. Hence, the “strictness” property of S2PL is preserved – the

locks are released only after the transaction had ended. This is different from standard two-phase locking (2PL), which also consist of two phases: acquiring locks without releasing any (Phase 1) and releasing locks without acquiring any (Phase 2), but it does not require that the Phase 2 happens only after transaction has ended.

Snapshot Isolation

Of particular interest to the work described in the thesis is the snapshot isolation. SI is commonly implemented using extensions of *multiversion mixed method* described in (Bernstein, Hadzilacos et al. 1987). A transaction executing in snapshot isolation operates on a snapshot of committed data, which is taken upon the transaction's *begin*. Snapshot isolation guarantees that all reads of a transaction see a consistent snapshot of the database. Additionally, any write performed during the transaction will be seen by subsequent reads within that same transaction. A transaction aborts only due to *write-write* conflicts when some of its operations try to modify data item(s) that had been updated by concurrent transactions. For a *begin* operation, b, and a *commit* operation, c, where b_i, $c_i \in T_i$ and $c_j \in T_j$, we say that the two transactions, T_i and T_j, are concurrent if the following holds:

$$b_i < c_j < c_i$$

The absence of conflicts between readers and writers in *snapshot isolation* improves performance and makes it more appealing than the traditional *serializable* isolation level. This is particularly evident in the workloads characterised with long-running read-only transactions and short modifying transactions.

In most real-world DBMSs the snapshot isolation is implemented using S2PL by acquiring exclusive locks for writing data items. Instead of waiting for transaction commit these concurrency control mechanisms check for *write-write* conflicts during transaction executions using *first-committer-wins* and *first-updater-wins rules* (Fekete, O'Neil et al. 2004).

Let us explain the mechanism in somewhat more detail. In order to write a data item *x* transaction T_i has to obtain the exclusive lock. There are two possibilities:

a) if the lock is available T_i performs a version check against the executions of the concurrent transactions. Two outcomes are likely: if a concurrent transaction had modified the same data item and it had already committed, T_i has to abort (*first-updater-wins*); otherwise it performs the operation.

b) in the case the lock is unavailable, because another transaction T_j has an exclusive access, T_i is blocked. If T_j commits, the version check results in aborting T_i (*first-committer-wins*). On contrary, if T_j aborts, the version check results in granting the lock to T_i so that it can subsequently proceed.

In both cases, a) and b), the key is that the version checks are performed at the same time T_i attempts to create a version. We call this *version-creation-time conflict check*.

An alternative to the particular use of S2PL is the possibility for transaction T_i to execute on the particular snapshot (taken at the time it starts) in the *private universe* (Fekete 2005). In this way, the transaction acquires the lock, performs version check (using first-committer-wins rule) and transfers the version from the private universe to the database only in the end of the transaction. We call this *commit-time conflict check*. This approach brings unnecessary delay because it postpones the validation of the updates until the end of transaction. In any case the locking is necessary for this approach too, as stated in (Fekete 2005):

"*While it is not mentioned in (Berenson, Bernstein et al. 1995), implementations of SI such as Oracle's ensure that a version of an item x produced by an SI transaction T must be protected by an exclusive lock from the time it leaves any private universe of T, until (and including the instant when) the version is installed because T commits*".

2.2.4. Liveness

Traditionally database replication protocols have been more concerned with safety than liveness properties. The latter is usually characterised with only blocking/non-blocking nature of a protocol – if a transaction eventually terminates, commits or aborts, the protocol is classified as a non-blocking one. This goes against the frequent concern of a database user, who would like to know if a transaction (eventually) commits. To reason about this matter we use the classification of liveness from (Pedone and Guerraoui 1997) where three liveness degrees are specified:

- *Liveness 3* (the highest degree) ensures that every transaction commits.
- *Liveness 2* ensures that read-only transactions are never aborted.
- *Liveness 1* ensures that every transaction eventually terminates (i.e. commits or aborts).

2.3. *Database Replication*

Database replication is a process of sharing data between redundant resources, which typically belong to a system of physically distributed nodes (commonly referred to as *replicas*). A replicated database system implements either a *full* replication (every node stores a copy of all data items) or a *partial* replication (each node has a subset of data items). Database replication is a thoroughly studied subject. Two main challenges of database replication are *concurrency control* and *replica control*. The former aims at isolating transactions with conflicting operations, while the latter ensures the consistency of data on the replicas. The work of (Gray, Helland et al. 1996) showed that replication solutions can be categorised according to the:

- Place *where* the writes take place.
- Time *when* the writes happen.

The first parameter divides the solutions into *primary copy* and *update everywhere* approaches. Primary copy approach designates only one replica to accept the writes. By contrast, in the update everywhere approach, writes are executed on all (available) replicas (Bernstein, Hadzilacos et al. 1987). The forwarding of updates to remote replicas incurs an overhead in the primary copy approach while the most common challenge in update everywhere replication is conflict resolution. The second parameter divides the solutions into *eager* and *lazy* replication. Eager solutions guarantee that the writes are propagated to all replicas before transaction *commit*. This has a negative impact on system performance, but ensures database consistency in a straightforward way. Lazy solutions perform writes after commit. They offer improved performance at the possible expense of compromising database consistency. If two transactions update different copies for the same data item with different values, data becomes inconsistent.

Another classification of database replication protocols identifies two broad groups: *middleware-based* and *kernel-based*. The protocols of the former group are easier to develop and can be maintained independently from the database servers they operate on. On the other hand, they are at a disadvantage because no access to the potentially useful concurrency control mechanism of the database server kernel is available. In this way concurrency control might have to be performed on a coarser level of granularity.

2.3.1. ROWAA-Based Replication

Eager replication protocols have been based on the *read-one/write-all* ROWA protocol (Bernstein, Hadzilacos et al. 1987). While read operations are executed only at one site in ROWA, updates are performed on all the replicas. However the main disadvantage of the protocol is its blocking nature, i.e. when a replica fails ROWA cannot continue. To remedy the deficiency, a refined version of the protocol was suggested, *read-once/write-all-available (ROWAA)*. In ROWAA replica failures are tolerated by updating only the available copies of data items.

As suggested in (Kemme 2000) one of the drawbacks of traditional ROWAA solutions is the message overhead. If the updates of a transaction are executed *immediately* on all replicas, an update message involves a request and an acknowledgement per each copy of data item. Clearly, this will have a significant impact on the scalability of this approach. It is also the case that aborting a transaction will cost less if the update has been executed only on a single replica than if the updates have been immediately propagated to all replicas. *Deferred* writing (Bernstein, Hadzilacos et al. 1987) was proposed as an alternative to immediate writing. All the writes are executed on one replica and at the end of a transaction they are bundled together in one message and sent to all other replicas. Deferred writing, however, exhibit an overhead because the commitment of each transaction will be delayed by possible large volume of writes to be executed. The execution of writes in the critical path is alleviated with the use of writesets – the modifications of a transaction are extracted, propagated and applied in a single unit instead of executing full SQL operations. This drawback does not exist when using immediate writing approach where processing of the writes happens in parallel. Another drawback of deferred writing is that detection of possible conflicts among transactions is delayed. While immediate writing might detect conflict during the execution of transactions, the conflict detection is performed at the end of transaction executions when deferred writing is the technique of choice.

A suite of replication protocols based on ROWAA has been offered as an alternative to the traditional *quorum* solutions (Kemme 2000). The underlying idea of quorum protocols is that read and write operations have to access a subset (quorum) of replicas. Each operation succeeds if the quorum agrees to execute it. It has been argued that, although quorums could decrease execution and communication overhead

their main disadvantage is that they do not scale well. In addition, complexity of read operations, commonly observed in many real life applications is better suited to ROWAA than quorums, since reads are done only on one replica in the case of the former. The work of Jimenez-Peris et al. (Jimenez-Peris, Patino-Martinez et al. 2003) provides a comparison of ROWAA and quorum solutions and indicates that the former is the replication protocol of choice for many types of applications. On the other hand Wool et al. (Wool 1998) argued that, due to the increasing speed of networks compared to hard disk drives, quorum solutions might be able to offer a way of scaling up throughput in heavily loaded environments.

2.3.2. Correctness in Replicated Databases

The strongest correctness criterion for replicated databases is *1-copy serializability (1SR)* (Bernstein, Hadzilacos et al. 1987). It represents an extension of the conflict-serializability defined for centralized databases (Section 2.2.3). It utilizes two types of histories for representing the correctness of replication protocols: Replicated Data (RD) histories and one-copy (1C) histories. The former characterizes the execution of operations on the replicated database and the latter characterizes the user's view of the replicated database as a single copy database - although a replicated database comprises multiple copies of data items, users perceive each data item as one logical copy. Most importantly, the criterion states that a replication protocol ensures 1SR if for any interleaved execution of transactions there is an equivalent serial execution of those transactions performed on the logical copy of the database. Testing the correctness of replication protocols has been traditionally performed using *replicated data serialisation graphs* (RDSG), which are extensions of SGs to replicated data. Similar to a history and the corresponding SG in a centralised database, a replicated data history that can be represented by an acyclic RDSG is 1-copy serializable (Theorem 8.5 from (Bernstein, Hadzilacos et al. 1987)). Additionally, *atomicity* of transactions guarantees that each transaction executes successfully (commits) on all, or at none of the replicas, even in the presence of failures. Different replica control solutions vary in level of isolation they offer and some of them violate the atomicity property.

Lin et al. (Lin, Kemme et al. 2005) defined criteria for correctness of replicated databases when each of the underlying replicas offers snapshot isolation. The correctness criterion, referred to as *1-copy snapshot isolation* (1-copy-SI), guarantees

that an execution of transactions over a set of replicas produces a global schedule that is equivalent to a schedule produced by a centralised database system which offers snapshot isolation. The authors provide the following three definitions to formalise 1-copy-SI correctness:

Definition 1 (SI-Schedule)*. Let $\mathcal{T}$ be a set of committed transactions, where each transaction T_i is defined by its readset RS_i and writeset WS_i. An SI-schedule S over $\mathcal{T}$ is a sequence of operations $o \in \{b, c\}$. Let $(o_i < o_j)$ denote that o_i occurs before o_j in S. S has the following properties.*

i. *For each $T_i \in \mathcal{T}$: $(b_i < c_i) \in S$.*
ii. *If $(b_i < c_j < c_i) \in S$, then $WS_i \cap WS_j = \{\}$.*

The read and write operations are excluded from *Definition 1* because the transaction boundary operations, begin (b) and commit (c), implicitly determine the logical time of their executions: a begin of transaction T_i indicates when its reads have taken place and similarly the commit of T_i indicates when the write operations take effect. This reasoning is based on the characteristics of SI (Section 2.2.2 and 2.2.3)

Definition 2 (SI-Equivalence)*. Let S^1 and S^2 be two SI-schedules over the same set of transactions $\mathcal{T}$. S^1 and S^2 are SI-equivalent if for* ***any*** *two transactions T_i, $T_j \in \mathcal{T}$ the following holds:*

i. *if $WS_i \cap WS_j \neq \{\}$: $(c_i < c_j) \in S^1 \Leftrightarrow (c_i < c_j) \in S^2$.*
ii. *if $WS_i \cap RS_j \neq \{\}$: $(c_i < b_j) \in S^1 \Leftrightarrow (c_i < b_j) \in S^2$.*

Definition 2 is based on the equivalence definitions as specified for the non-replicated database systems using serializability theory. Condition *i.* ensures that the order of committed transactions with overlapping writesets is the same in both schedules. Thus, the final writes (a write performed by a committed transaction after which no other committed transaction modified the same data item) are the same in the two schedules and each *prefix* of the partial order of committed transactions in both schedules is an SI schedule. Condition *ii.* ensures that if in one schedule a transaction, T_j, reads data modified by a committed transaction, T_i, the same will be true for the other schedule – the begin of T_j will follow the commit of T_i. The condition, on the other hand, does not specify which transaction exactly T_i reads from in either of the schedules.

In order to define 1-copy-SI criterion the authors of (Lin, Kemme et al. 2005) assume the following:

- Each replica produces SI schedules.
- Replication is based on ROWA approach: each transaction is executed on a local replica and only its writes are propagated to the remaining ones. To formalise the ROWA approach the authors use a mapper function *rmap*. The input to the function is a set of transactions $\mathcal{T}$ and a set of replicas $\mathcal{R}$. Each update transaction is transformed into a set of transactions $\{T_i^k | R^k \in \mathcal{R}\}$, one for each replica. Only one of these transformed transactions contains both, the read and the write set of the original transaction - this is the *local* transaction. The rest of the transactions are remote and consist of only the writeset of the transaction. Every read transaction, on the other hand, has a single transformation into a local transaction.

Definition 3 (1-Copy-SI). *Let $\mathcal{R}$ be a set of replicas following ROWA approach. Let $\mathcal{T}$ be a set of submitted transactions for which $T_i \in \mathcal{T}$ committed at its local site. Let S^k be the SI-schedule over the set of committed transactions $\mathcal{T}^k$ at replica $R^k \in \mathcal{R}$. Then $\mathcal{R}$ ensures 1-copy-SI if the following is true:*

i. *There is ROWA mapper function,* rmap, *such that* $\bigcup_k \mathcal{T}^k = \text{rmap}(\mathcal{T}, \mathcal{R})$
ii. *There is an SI-schedule S over $\mathcal{T}$ such that for each S^k and $T_i^k, T_j^k \in \mathcal{T}^k$ being transformations of $T_i, T_j \in \mathcal{T}$:*
    a. *if* $WS_i^k \cap WS_j^k \neq \{\} : (c_i^k < c_j^k) \in S^k \Leftrightarrow (c_i < c_j) \in S$,
    b. *if* $WS_i^k \cap RS_j^k \neq \{\} : (c_i^k < b_j^k) \in S^k \Leftrightarrow (c_i < b_j) \in S$.

From the condition *i.,* we infer an existence of an *rmap* function that maps committed transactions as a subset of the set of submitted ones. Condition *ii.* ensures equivalence between a schedule produced by a replica, S^k, and the global schedule, S, over the set of all transactions $\mathcal{T}$. Due to the use of ROWA approach, the definition of equivalence as stated in *Definition 2* has to be modified. The condition *i.* from the *Definition 2* holds between every S^k and S for all committed transactions, because the writes are executed on all replicas. However, the reads-from relation of a schedule S^k is the same as in S (condition *ii.* from *Definition 2*) for only the subset of the readsets obtained at the replica R^k. There are two consequences of the 1-copy-SI definition:

- The position of the begin operations of remote transactions is arbitrary since they do not include read operations.

- The position of the commits of the read-only transactions is arbitrary since they do not include any write operations.

Similarly to 1-copy-SI, Elnikety et al. (Elnikety, Zwaenepoel et al. 2005) defined Generalised Snapshot Isolation (GSI) - a correctness criterion for replicated databases that offer snapshot isolation. GSI is an extension to the snapshot isolation as found in centralized databases. The authors formalize the "centralized" snapshot isolation and refer to it as Conventional Snapshot Isolation (CSI). To model the timing relationships between transactions the following definitions of the operations in a transaction, T_i, are given:

- *snapshot*(T_i) – the time when T_i's snapshot is taken.
- *start*(T_i) – the time of the first operation of T_i.
- *commit*(T_i) – the time of commit of T_i.
- *abort*(T_i) – the time of abort of T_i.

In addition, they showed that serializability can be guaranteed under GSI by ensuring that either a static property, which can be checked by examining the transactional profile, or a dynamic one, which checks the intersection between the readsets and writesets of overlapping transactions, is satisfied.

The definitions of GSI and CSI, and the corresponding definitions of *impacting* transactions, are as follows:

Generalised Snapshot Isolation (GSI) Definition:

- *G1. (GSI Read Rule)*
  $\forall\ T_i, X_j$ *such that* $R_i(X_j) \in h$:
  1. $W_j(X_j) \in h$ **and** $C_j \in h$;
  2. $commit(T_j) < snapshot(T_i)$;
  3. $\forall\ T_k$ *such that* $W_k(X_k), C_k \in h$:
     $commit(T_k) < commit(T_j)$ **or** $snapshot(T_i) < commit(T_k)$;
- *G2. (GSI Commit Rule)*
  $\forall\ T_i, T_j$ *such that* $C_i, C_j \in h$:
  1. $\neg (T_j$ *impacts* $T_i)$

Definition of Impacting Transactions for GSI:

- <u>T_i *impacts* T_j ***iff:***</u>

$snapshot(T_i) < commit(T_j) < commit(T_i)$ **and** $writeset(T_i) \cap writeset(T_j) \neq \{\}$

Conventional Snapshot Isolation (CSI) Definition:

- *C1. (CSI Read Rule)*

$\forall\ T_i, X_j$ *such that* $R_i(X_j) \in h$:

1. $W_j(X_j) \in h$ **and** $C_j \in h$;
2. $commit(T_j) < snapshot(T_i)$;
3. $\forall\ T_k$ *such that* $W_k(X_k), C_k \in h$:

   $commit(T_k) < commit(T_j)$ **or** $start(T_i) < commit(T_k)$;

- *C2. (CSI Commit Rule)*
  $\forall\ T_i, T_j$ *such that* $C_i, C_j \in h$:
  1. $\neg (T_j$ *impacts* $T_i)$

Definition of Impacting Transactions for CSI:

- T_j *impacts* T_i ***iff:***
$start(T_i) < commit(T_j) < commit(T_i)$ **and** $writeset(T_i) \cap writeset(T_j) \neq \{\}$

CSI states that the *last* snapshot, committed on any of the database replicas, in respect to the transaction start time, is available. CSI is a special case of GSI as the latter does not specify which database snapshot should a transaction observe, i.e. $snapshot(T_i) = start(T_i)$ in CSI. We aid the description of the difference between GSI and CSI with the following example. Let history $h = W_i(X_i), C_i, W_j(X_j), C_j, R_k(X_i), W_k(Y_k), C_k$. The history is not permitted by CSI because T_k reads an "old" snapshot, $snapshot(T_i)$, instead of the *last* one, $snapshot(T_j)$. However h is a GSI history since $snapshot(T_k) = commit(T_i)$ is allowed.

2.3.3. Conflicts and Deadlocks

As detailed in (Gray, Helland et al. 1996), a severe drawback of eager, update-everywhere replication is the high conflict rate and probability of deadlocks. The work showed that, in some situations, the probability of deadlocks is directly proportional to n^3, where n is the number of replicas. The observation is not surprising: as the number of replicas increases, the time to lock the resources will increase too and transaction execution times will deteriorate. Furthermore, the longer transaction times are caused by the additional communication overhead when the number of replicas increases. Certain database replication solutions are prone to a *distributed deadlock*, a more complicated form of deadlock. This happens if a resource lock is acquired in different order on different replicas. Figure 2-1 shows two concurrent transactions T_1 and T_2, which are competing for resource A while executing in a replicated database system with two replicas R_x and R_y. The order of

lock requests by T_1 and T_2 is different on the two replicas, i.e. T_2 is blocked waiting for T_1 on R_x and vice versa is true on R_y. As a result, in replication schemes where a transaction commits only after all the replicas are ready to do so, the transactions would be deadlocked without possibility to progress further.

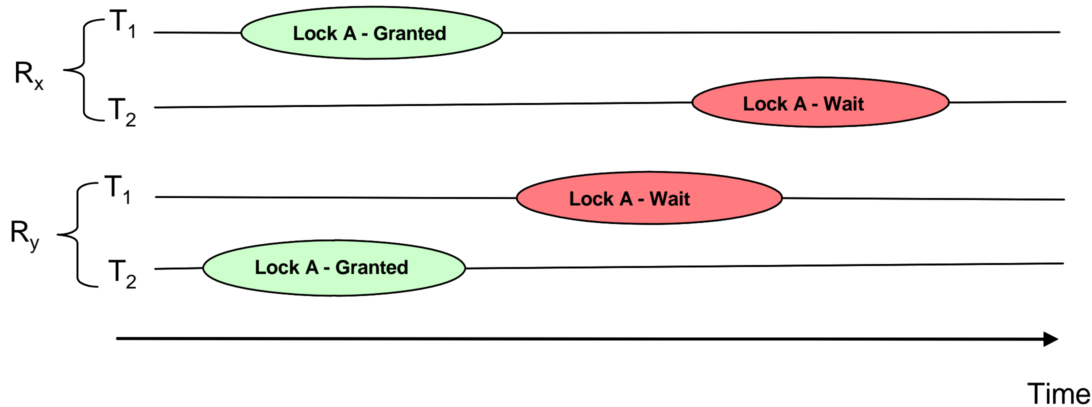

Figure 2-1 An example of distributed deadlock.

It has been suggested that *group communication systems* (Hadzilacos and Toueg 1993) be used as a means of reducing conflicts and avoiding deadlocks as well as ensuring consistent data on multiple replicas. These systems are capable of ensuring that a message multicast in a group will be delivered in the same total order on all group members. This holds for the sender of the message too. Many replication protocols, e.g. (Agrawal, Alonso et al. 1997), (Kemme and Alonso 2000a), combine group communication primitives with deferred updates technique, in which, usually, the write operations of a transaction are executed on one replica, grouped in one message (*writeset*) and delivered to all the replicas in the same total order.

2.3.4. Transaction Atomicity

In replicated databases atomicity of a transaction has to be guaranteed - it is necessary to make sure that all replicas terminate the transaction in the same way, i.e. all replicas either commit or abort a transaction. As described in (Weismann, Pedone et al. 2000) there are two techniques to ensure transaction atomicity in a replicated database system: *voting* and *non-voting* techniques. Voting techniques, traditionally, use atomic commitment protocols to terminate transactions. A well-known variant of the atomic commitment protocol is the *2-Phase Commit (2PC)* protocol (Skeen 1981). The phases of a 2PC protocol could be summarised as follows:

- A *coordinator* sends a request to each replica for a vote (to commit or abort).
- Upon receiving the request each replica replies with a message, YES (commit) or NO (abort). If the vote is NO the replica aborts the execution.
- Upon collecting all the votes the coordinator decides on the outcome of the transaction:
  - **If all** replicas have voted YES, the coordinator notifies the replicas to commit.
  - **If a** replica voted NO, the coordinator notifies the replicas to abort.
- Upon receiving the notification (commit or abort) from the coordinator, a replica decides accordingly.

2PC is a blocking implementation of a more general atomic commitment solution. Atomic commitment protocols suffer long transaction times because the synchronisation point is dependant on the performance of all the replicas. It is the slowest server that determines the response time of a transaction.

In non-voting techniques each server must independently decide on the same serialisation order among transactions. For example, when using group communication primitives, total order detects possible conflicts between transactions and warrants the same serialisation order on all replicas. Since an abort may happen due to different reasons in a database, e.g. due to an interaction with a local operation or a violation of a consistency constraint, additional measures have to be taken to guarantee atomic termination of a transaction. To that end a *certification* step (Pedone, Guerraoui et al. 2003) has to be performed or a commit order has to be decided by sending an additional message to all replicas, like in SER-D protocol (Kemme 2000). Using non-voting techniques each replica is allowed to terminate a transaction without waiting for all other replicas to finish. This is a performance improvement towards shorter response times.

2.4. TPC-C – an On-Line Transaction Processing Benchmark

In order to evaluate performance implications of using diverse redundancy we used our own implementation of the industry-standard benchmark for online transaction processing - TPC-C (TPC 2002a). A *real* workload, the one being observed on a system under normal operations in true environment, is preferred choice in performance studies (Jain 1991). However, these workloads are not repeatable and

therefore not suitable as test workloads. Consequently, we chose TPC-C, a *synthetic* benchmark, as the basis for our performance evaluation. Its representativeness of an order-entry system and wide adoption in industry and academia warrants the choice. TPC-C defines five types of transactions: *New-Order* (NO), *Payment* (P), *Order-Status* (OS), *Delivery* (D) and *Stock-Level* (SL) and sets the probability of execution of each. NO, P and D are update transactions with a different number of read and write operations while SL and OS are read-only transactions consisting of only read operations. In our implementation of TPC-C, OS transaction consists of three and SL of two SELECT operations. NO and P transactions are the most frequently executed ones. The minimum probability of execution for each transaction type is as follows: P – 43%, OS – 4%, D – 4% and SL – 4%. TPC-C does not specify the minimum frequency for NO transactions because the throughput measure of interest is the number of NO transactions per minute (under the specified mix of transaction types). The benchmark provides for performance comparison of the database servers from different vendors, with different hardware configurations and operating systems.

The workload of TPC-C is highly write-intensive and in particular the application is limited with a random I/O access pattern. This is why it is not unusual that commercial TPC-C results report the use of several CPU cores with multiple RAID controllers and hundreds, or even thousands, of disk drives (Hewlett-Packard 2005). One relation, *warehouse*, is used as the base unit of scaling, e.g. the standard specifies that the number of clients is 10 times greater than the number of warehouses. To model the time spent by interactive clients on interpreting results, the standard introduces *think times*, with an additional consequence of reducing conflict rates, e.g. a conflict happens when multiple clients modify the same warehouse record. Greater *think times* decrease the contention and load on the system by increasing time breaks between transactions requested by a particular client. The values of the mean *think times* for each transaction type are as follows: NO – 12 sec, P – 12 sec, OS – 10 sec, D – 5 sec and SL – 5 sec.

The standard defines the upper bounds on the 90th percentile for each transaction type and the proposed constraints are as follows: SL – 20 sec and 5 sec for all other transaction types. In the case of Delivery transaction, the specified value is pertinent to the interactive part only, while the value of 80 sec is specified for the whole transaction, including the deferred portion. The fixed upper bounds for 90th percentile response time are set based on the assumption that the utilization level of the

hardware is near 100%. The choice for the values used as response time constraints was made by vote at TPC. Several values were proposed and the values that obtained a majority agreement were included in the standard benchmark. These values are somewhat loose, even 10 years ago when they were set, and could easily be changed to something shorter (Raab 2005). These values are inherently dependant on the characteristic of the system under test (SUT), such as hardware and database size as well as the load and the profile of the experiments.

The access pattern of TPC-C benchmark is characterised with a high degree of locality when the warehouse table is accessed. Multiple clients read and modify a single row in the table. While this access hotspot might be beneficial in terms of keeping data in memory, it has a significant disadvantage of generating frequent conflicts and incurs a performance cost as a result of resolving these conflicts. This is exacerbated in workloads with slow-running transactions that cause the conflicts to persist for long periods of time.

Our implementation of the TPC-C did not require the use of any proprietary features from any of the servers. The SQL operations were implemented using the common subset of SQL.

3. Architecture of DivRep Middleware

A pessimist is an optimist with experience

Anonymous

3.1. DivRep – Replication with Diverse Database Servers

We have developed a middleware-based database replication solution, *DivRep*, which uses redundant database servers as the underlying components. Although non-diverse database servers can be deployed, we propose that DivRep is used to build a fault-tolerant server (*FT-node*) employing diverse redundancy. Figure 3-1 depicts the architecture of an FT-node with two diverse replicas. DivRep supports interactive transactions and replication is done at the SQL operation level. Hence, dependence between SQL operations within a transaction is allowed, a feature unavailable in many replication solutions, e.g. in some replication solutions clients are assumed to submit a whole transaction at once and as a consequence SQL operations created by an operator on the fly, possibly using results of previous operations, are impossible to handle. The middleware propagates the operations generated by the client applications to both diverse replicas for execution. The results from the replicas are collected by the middleware, and in the case of a positive adjudication, the middleware reports a result back to the client application(s). In this way the parallel-redundant architecture using two diverse DBMS products provides high error detection rate (Section 2.1) via comparison of results. Clearly, this architecture differs from ROWAA scheme. In the new architecture both diverse replicas execute all operations (including the reads), while in ROWAA scheme all active replicas execute only the writes submitted to any replica and the respective local read operations.

The overall performance of the system that is shown in Figure 3-1 depends on the performance of the diverse replicas deployed and on the performance characteristics of the middleware itself. For instance, the middleware can use different adjudication mechanisms. A few reasonable alternatives are listed below:

- *Slowest response regime*. The middleware collects the results of the *individual SQL operations* (a multitude of which constitute a whole transaction) executed by the

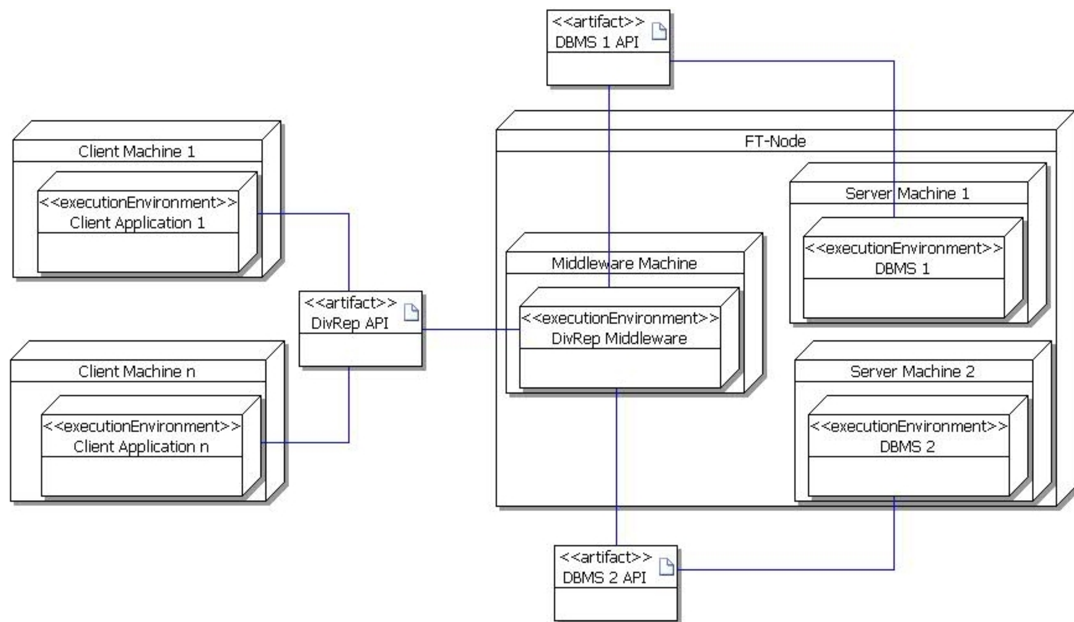

Figure 3-1 Fault-tolerant server node (FT-node) with two diverse DBMSs (DBMS 1 and DBMS 2) as a UML 2.0 deployment diagram. The middleware "hides" the servers from the clients (1 to n) for which the data storage appears as a single DBMS.

diverse replicas. Once a sufficient number of responses are collected, they are adjudicated and only if identical responses from both replicas are observed a successful completion of the operation is reported back to the client application. Subsequently the client sends the following operation to the middleware for processing.

- *Pessimistic response regime.* Alternatively, the middleware may buffer the operations coming from a client application and make them available to the diverse replicas as soon as the operations are placed in the respective buffers. Each diverse replica collects the next available operation from its respective buffer, executes it, marks it as being completed and makes the response from the operation available to the middleware. As soon as the middleware receives the first response to an operation from a replica, it is immediately passed on to the client application, thus letting the client application proceed with the other operations within the transaction. The fastest response comes from either of the DBMSs, depending on the SQL operation (Figure 3-2). Responses from the diverse replicas to the same operation are adjudicated later, after both replicas produce them, but before the end of the transaction (the condition of *eager* replication is satisfied). Buffering the operations in the middleware allows the diverse replicas to work at a maximum speed within transactions, as shown in Figure 3-2 (DBMS 1 would start execution of the next SQL operation even though the DBMS 2 has not finished the previous

one, as indicated with the dashed rectangle). The transactions are committed (or aborted) based on the outcome of adjudicating the results of the operations. Commit is only applied if the replicas execute all operations successfully and all responses are positively adjudicated. Otherwise, the transaction is aborted.

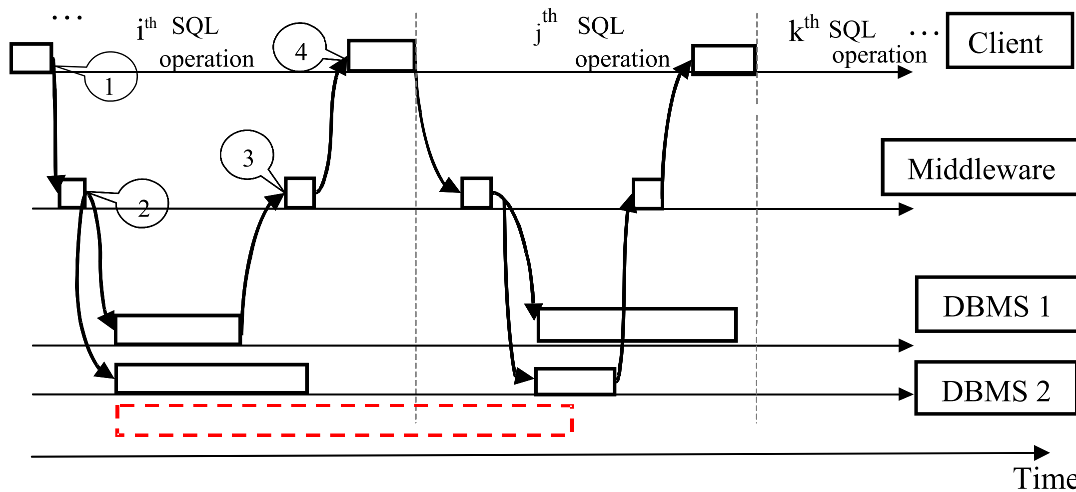

Figure 3-2 Timing diagram of a client communicating with two, possibly diverse, database servers and the middleware running in the *pessimistic* or the *optimistic* response regime. The meanings of the callouts are: 1 – the client sends an SQL operation to the middleware; 2 – the middleware translates the request to the dialects of the servers and places the resulting SQL operations, or sequences of SQL operations, in the respective server buffers; 3 – the fastest response is received by the middleware; 4 - the middleware sends the response to the client. Processing of only a subset of SQL operations in a transaction is depicted. The dashed rectangle shows an alternative effect of asynchronous execution by the two DBMSs - it indicates that DBMS 2 will not be ready to start j^{th} SQL operation at the same time with DBMS 1.

- *Optimistic response regime*. This is similar to the *pessimistic* response except:
  - *No adjudication* of the responses from the diverse replicas is applied.
  - A *skip* feature (Figure 3-3) is implemented in the middleware as follows. Before a replica, e.g. DBMS 1, executes a *read* (i.e. SELECT) operation it checks if a response to this operation has already been received from the other replica, DBMS 2. If so then DBMS 1 does not execute the operation (i.e. skips it). The modifying SQL operations (DELETE, INSERT and UPDATE) are executed on all servers, i.e. they cannot be skipped. The functionality of looking up the next operation and the *skip* feature is, of course, implemented in the middleware, which relays to the DBMSs the operations for execution. If a read operation is to be skipped, then the middleware simply does not pass it to the respective DBMS for execution. Clearly, this regime of operation does not offer the same level of protection as the previous ones. It may, however, be adequate in many cases.

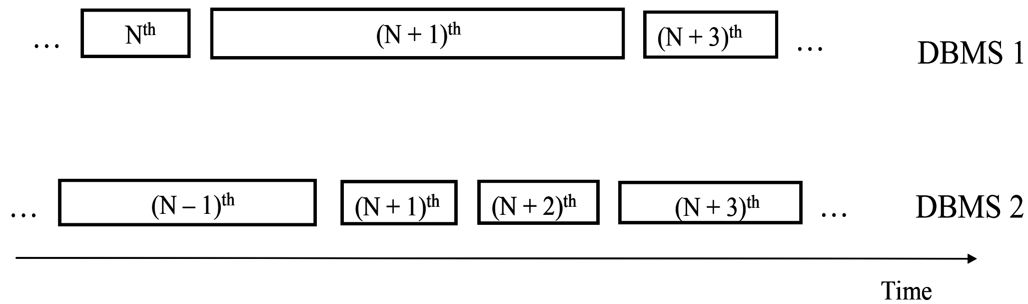

Figure 3-3 Implications of using *skip* feature in the *optimistic* regime of operation. Let us assume that two replicas, DBMS 1 and DBMS 2, are executing a read-only transaction and only a subset of read operations is depicted. Due to the variable duration of the reads (represented by the corresponding rectangles) the load on each replica will be decreased: DBMS 1 would omit the execution of $(N+2)^{th}$ read and similarly DBMS 2 would skip the Nth read operation.

There might exist systematic differences between the times it takes diverse DBMSs to execute the same operation. This may be due to, for example, the respective execution plans being different, the concurrency control mechanisms being implemented differently, etc. When the *slowest response* regime is used, such differences will inevitably lead to the FT-node being slower than the respective DBMSs it consists of. The *slowest response* regime incurs an unnecessary performance penalty: the processing of client's requests is suspended until all the replicas have produced the results for a particular operation and the adjudication has completed. When the *optimistic* regime is used, however, the systematic difference might lead to improved performance. If the mix of operations within a transaction is such that both servers 'skip' operations, then the transaction might take the FT-node less time than either of the DBMSs it uses. In the *pessimistic* response regime the *skip* feature is not used and the best that the FT-node can do during transaction execution is to process SQL operations as fast as the faster of the two servers can. However, the FT-Node waits for both servers to complete *all* operations and performs adjudication, and thus diversity cannot bring any performance gains. Due to the synchronous execution of SQL operations and result adjudication in the *slowest response* regime, DivRep is primarily concerned with the remaining two regimes of operation: *pessimistic* and *optimistic* response regimes.

We propose that *a pair of* replicas is deployed within an FT-node for improved detection of non-crash failures (Gashi, Popov et al. 2007), (Gashi 2007). Should a higher level of replication be required, e.g. for better scalability, then the FT-node can be combined with another database replication scheme, which is considered adequate

for a particular set of requirements. These can be schemes for *eager* database replication, e.g. based on group communication primitives, or even *lazy* replication. In either case the FT-node will replace a replica of a particular DBMS used by the particular database replication scheme (Figure 3-4).

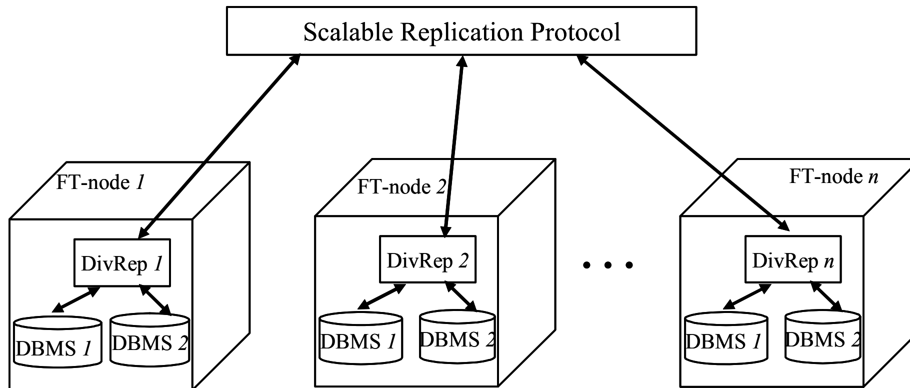

Figure 3-4 A scalable replication protocol (SRP) deploying FT-nodes as underlying replicas instead of using non-diverse replicas as in its original configuration. Error detection is provided with FT-nodes (*1* to *n*) – each one deploys two diverse servers (DBMS *1* and DBMS *2*). High-performance and scalability are provided by the SRP. Database consistency is provided via the interaction of DivRep and the SRP.

Whichever regime the FT-node operates under, data consistency between the diverse replicas must be guaranteed, which is typically defined as *1-copy serialisability* (Bernstein, Hadzilacos et al. 1987), although recently definitions of correctness when DBMSs offer snapshot isolation have been specified, (Lin, Kemme et al. 2005), (Elnikety, Zwaenepoel et al. 2005) (Section 2.3.2).

3.1.1. Dependable Replication Algorithm (DRA)

The choice of a particular replica control mechanism is, to certain extent, dependant on the *isolation level* provided by the underlying database servers (Sections 2.2.2 and 2.3.2). In our solution we propose a mixed mode for concurrency control, where we combine the mechanisms of the database servers with the additional replica control performed by the middleware. Concurrency control mechanisms of database systems are effective, yet they use complex algorithms and introduce specific features. When designing the replication algorithm we wanted to reuse the effectiveness of these algorithms, but at the same time add least complexity and assure DRA is independent from the specifics of different servers.

We assume that the underlying database servers offer *snapshot isolation*, thus only concurrent writes of the same data item cause conflicts. In addition we assume that the

replicas, instead of waiting for transaction commit, check for *write-write* conflicts during transaction executions – they perform *version-creation-time conflict check* (see Snapshot Isolation in 2.2.3).

We have implemented a database replication algorithm*, Dependable Replication Algorithm* (*DRA)*, which guarantees database consistency when DivRep middleware is employed. The algorithm guarantees that the replicated system produces transaction execution histories equivalent to a history of a centralized SI scheduler, thus it guarantees *1-copy Snapshot Isolation* (*1-Copy-SI*) (Lin, Kemme et al. 2005). 1-copy-SI is based on the traditional serializability theory and it describes the correctness criteria for transaction executions in a replicated system consisting of SI-compliant servers. Moreover, DRA preserves *conventional snapshot isolation* as described in (Elnikety, Zwaenepoel et al. 2005) and similarly *strong SI* as explained in (Daudjee and Salem 2006). Analogous to the regimes of operation of DivRep middleware there are two variants of DRA, which implement the *pessimistic* and the *optimistic* regimes of operation.

Figure 3-5 presents a pseudo-code of the *DRA* algorithm executing in the *pessimistic* regime of operation. It describes an execution of a transaction submitted by a particular client; multiple such executions are taking place concurrently at DivRep – each client’s requests are served by a dedicated DivRep’s thread. Both DML (Data Manipulation Language) operations (SELECT, DELETE, INSERT and UPDATE) and transaction boundary operations (*begins*, *commits* and *aborts*) are submitted to the middleware. The middleware, i.e. the dedicated execution context for a particular client, forwards the operations to the replicas. The communication with each replica deployed in the FT-node is performed by a separate thread of execution. Replicas communicate with the middleware but not with each other. The most up-to-date versions of data are stored on every replica.

Managing DML Operations

Upon receiving a *read* or a *write* request from a client the middleware forwards it to the deployed replicas. As soon as the fastest response is received, the middleware returns it to the client. The client then sends the following request, possibly using the results of the previous response. If a *write-write* conflict is signalled by a replica, upon an attempt to execute a write operation, an exception is raised and the middleware records a “vote” for abort. Recall from Snapshot Isolation subsection in

```
1) Upon SQL Operation OP from Ti

   A) if OP is the begin then
      I)     obtain tb_mutex
      II)    begin Ti on both replicas /*create snapshot for Ti*/
      III)   release tb_mutex
      IV)    return to client /*return control*/

   B) else if OP is a read operation
      I)     send the read to both replicas
      II)    receive fastest response
      III)   return to client /*return control and the response*/

   C) else if OP is a write operation
      I)     send the write to both replicas
      II)    if a write-write conflict reported then
         (i)     set transaction abort /*vote for abort*/
      III)   else
         (i)     generate control read and send it to both replicas
         (ii)    receive fastest response for the write
         (iii)   return to client /*return control */

   D) else if OP is an abort /*client sends the abort operation*/
      I)     abort Ti on both replicas

   E) else /*it is a commit operation - use 2PC-DR*/
      I)     Upon both replicas and the Comparator voted
         (i)     if (an abort vote) /*a write-write conflict and/or a result comparison
                                    failed*/
               (a) abort Ti on both replicas
         (ii)    else /*both ready to commit and no mismatch between the results*/
               (a) obtain tb_mutex
               (b) commit Ti on both replicas
               (c) release tb_mutex


Comparator Function
1) Upon all results of a read or all results of a control-read operation ready
   A) Compare the results from both replicas
   B) If a mismatch is found then
      I)     set transaction abort /*vote for abort*/
```

Figure 3-5 Pseudo-code of DRA algorithm when DivRep middleware operates in the *pessimistic* regime.

2.2.3 that when the *first-committer-wins* rule is not enforced as part of a *version-creation-time conflict check*, a *write-write* conflict could be reported only once after a commit request (*commit-time conflict check*). If this was the case in DivRep, a replica would not be reporting the *write-write* exception at line 1.C.II., but after 1.E.I.ii.b (Figure 3-5). As a result an abort would have to be initiated, and the replicas would diverge had the commit already been executed on the other replica. Clearly the replication algorithm would have to change to prevent such inconsistencies. As mentioned previously, however, databases that offer snapshot isolation enforce the *first-committer-wins* rule as part of the write operation execution. Furthermore, without the specific application of the rule the abort rate might be higher because transactions would last longer due to the conflicts being detected in the end of a transaction.

The middleware compares the results (*Comparator Function*) of already executed SQL operations in parallel with forwarding client requests to replicas. The results of SELECT operations are compared in a straightforward way i.e. using the respective result sets returned by the replicas. After checking that the same number of rows is retrieved, an exhaustive value-for-value comparison is performed between the results from different replicas. If a particular order has not been specified in the query (using an ORDER BY clause) it is possible that the values will be ordered differently in the result sets. The comparison algorithm of DivRep makes sure that no inconsistency is reported only due to different ordering in the result sets.

In order to compare the effects of *write* operations the middleware generates *control read* operations. A *control read* is constructed by parsing the respective write operation (DELETE, INSERT or UPDATE), so that it queries the rows modified by the write. A control read is executed immediately after the *write* to retrieve the modified records. Under snapshot isolation each transaction observes its own writes and thus each control read operation captures the modification performed by the preceding write operation. Clearly, it is necessary for the comparison that the same state of the replicas is maintained for each transaction execution. Since the underlying database servers offer snapshot isolation and execution order of transaction boundaries is the same on both servers, the algorithm provides the necessary replica determinism. If no failure occurs the replicas produce the same results of SQL operations. The comparison of the results has to be performed before transaction commits. In this way the detection rate of server failures is increased and further,

well-known, mechanism for improved fault-tolerance (e.g. error containment, diagnosis and correction) can be performed. Applying these techniques to database replication using diverse redundancy is discussed in (Gashi, Popov et al. 2007) and (Gashi 2007). The use of the comparator function can help DRA cope with non-deterministic operations, a feature rarely guaranteed by most replication solutions. For example, execution of a non-deterministic function that returns the current date on two replicas can produce, legitimately, different results and the inconsistency can be detected and appropriately corrected using DRA. DivRep middleware could, for instance, calculate the value and rephrase the SQL operation or, on the other hand the use of non-deterministic functions could be simply avoided in the client application. A performance overhead is imposed when using the comparator function because the *control reads* force more round-trips, additional communication overhead, between the middleware and the replicas. Also the processing time of the comparison might be significant if large data sets are compared. These issues can be alleviated using different approaches. Using SQL extensions for *data-change* operations (DELETE, INSERT, UPDATE) (Behm, Rielau et al. 2004), which return a result set containing modified rows, would eliminate this overhead. This type of SQL operations is offered by DB2 database engine (IBM 2007). Using the capabilities of *data-change* operations, application developers can:

"… retrieve a result set containing the final column values of all rows that are updated or inserted. This is particularly useful for operations against tables with automatically generated columns, columns with default values, or columns whose values are altered by BEFORE triggers. Similarly, a result set containing the old values of updated or deleted rows can be retrieved" (IBM 2007).

An alternative option is to use *after* triggers (PostgreSQL 2007), (Borland 1999), (ISO 2003) for extraction of modified rows once DELETE, INSERT or UPDATE operation is executed. Either statement level or per-row triggers could be used. Another possibility, orthogonal to the use of control-reads and triggers, can be used for performance improvement. By hashing the results, of control reads or triggers, the results' comparison processing may be reduced. The network overhead might be minimised if instead of potentially large results only hash codes, generated by each replica, are sent to the middleware for comparison. However, to avoid the difference in hash functions of diverse replicas, the results could be propagated to the middleware, which then generates hash codes for comparison. In this case the network

overhead would not be eliminated (it is usually insignificant in LANs), while the result comparison processing would be improved. Hash collisions (when two distinct inputs into a hash function produce identical outputs) could possibly compromise the results' comparison. This would happen when the results from two replicas to the same operation are inconsistent but their hash values are the same. Nevertheless, these events are usually rare: hash collisions are rare per se (at least in well-designed functions) and it is unlikely that two databases produce different results with the same hash value. The aim of DivRep middleware to improve error detection can be undermined if its own fault-tolerant feature, result comparison, produces faulty responses. Namely it is possible that either control reads or triggers generate different results; but, in this case, the algorithm behaves the same as if the corresponding *writes* have produced divergent results – it reports inconsistency of replicas' results. Similarly the comparator function might be faulty – if it falsely declares that different results from diverse replicas are consistent the states of the databases might diverge. Due to its simplicity, though, we could assume perfect correctness of the comparator function with high confidence.

Managing Transaction Boundary Operations

When a client indicates the *begin* of a transaction T_i the corresponding middleware thread obtains *tb_mutex*, a global mutex for which all the middleware threads serving different clients are contending, and starts the transaction on the replicas. No boundary operation, *commit* or *begin*, from any other transaction T_j can execute while T_i holds *tb_mutex*. Therefore, the middleware admits execution of only one transaction boundary at the time, i.e. an overlap in execution of boundary operations from different transactions is prevented. This imposes total order on transaction boundary operations guaranteeing that the identical schedule of *commits* and *begins* is applied to both replicas. Atomic execution of boundary operations guarantees that a transaction operates on the equivalent snapshot of data on both replicas.

DRA uses an *atomic commitment* building block to handle *commit* operation received from the client. A variant of Two-Phase Commit protocol (2PC) (Gray 1978), (Skeen 1981) is used. The implementation is denoted as *2PC-DR* in the rest of the document. It represents a *blocking* implementation of the general atomic commitment problem and as such satisfies the following properties:

- *Agreement*
  - No two replicas and the Comparator *decide* different values (*abort* or *commit*). See 1.E.I.i.a, (*"abort* T_i on **both** replicas"), and 1.E.I.ii.b ("*commit* T_i on **both** replicas") in Figure 3-5.
- *Validity*
  - If a replica or the Comparator *votes abort* then abort is the only possible decision value. See 1.E.I.i in Figure 3-5 – it follows from 1.C.II.i in the algorithm and 1.B.I in the Comparator function.
  - If both replicas and the Comparator vote commit, then commit is the only possible decision value, see 1.E.I.ii in Figure 3-5.
- *Weak Termination*
  - If there are no crash failures then both replicas and the Comparator eventually decide.

The *strong termination* property, which satisfies the following: "All correct processes eventually decide" is unattainable with 2PC-DR. This does not come as a surprise since in general it is impossible to achieve strong termination in asynchronous systems subject to crash failures (Guerraoui 1995).

Once the replicas and the *Comparator* have "voted", the middleware ends the transaction on both replicas. A replica either successfully completes all operations in a transaction or it raises an exception. The middleware regards the former as a "vote" for the *commit* and the latter as a "vote" for the *abort*. Similarly, only if the Comparator reports no inconsistencies between the results a *commit* "vote" is recorded. In this way the time complexity of 2PC-DR is represented with two rounds (Bernstein, Hadzilacos et al. 1987). In the first round the participants, i.e. replicas, send their votes to the coordinator, i.e. middleware, and in the second the coordinator broadcasts the decision. There is no explicit request for voting (VOTE-REQ) sent by the coordinator. Hence, in the absence of replica or communication failures, the message complexity in 2PC-DR is *2n*; in each phase there are *n* messages exchanged.

The execution of *commits* is similar to the execution of *begins* - concurrent execution of the transaction boundary operations is prohibited. In order to minimise the performance overhead DRA broadcasts an abort message to the replicas as soon as one is reported by any of the replicas, instead of performing *agreement* phase once the replicas have finished the SQL operations and the comparison of results has been completed. This behaviour is similar to first-committer-wins rule: as soon as the *destiny* of a transaction is known on a replica (a *write-write* conflict is reported) the remaining one and the Comparator are notified, and the transaction is aborted. In this way transaction duration is shorter and the resource utilisation on replica machines is

more efficient. DivRep notifies the client that the transaction has been aborted. Subsequently the client repeats the transaction and the competition for the resources is reinitiated. Alternatively, DivRep could notify the client that a conflict has occurred and only subsequently, after the client has sent the abort message, end the transaction. However, since the conflict had been raised, the faith of the transaction is decided and thus it would be wasteful to first notify the client and only then abort the transaction – the locks on the resources would have been kept unnecessarily long and the transaction latency would increase. Application level aborts are conducted in a transparent manner – as soon as a client sends an abort DivRep ends the transactions on both replicas and notifies the client. DRA does not serialise the execution of the aborts, i.e. it does not obtain *tb_mutex*, because the changes created by the transaction will be invalidated and no snapshot will be created. The contention for *tb_mutex* is decreased in this way and thus shorter response times are observed.

Although DRA enables non-synchronised processing of SQL operations inside a transaction (the replicas execute SQL operations at their own pace), its performance is limited due to transaction boundaries being ordered serially (by synchronising *begins* and *commits*). This holds even in the cases when transactions do not have overlapping *writesets*. Nonetheless, serialisation of transactions and atomic commitment are needed to ensure the "latest" database snapshot (Elnikety, Zwaenepoel et al. 2005) has been installed on both replicas. In this way DRA guarantees that a transaction T obtains a particular snapshot – the snapshot of the database which reflects the writes of all transactions which committed globally (on both replicas) before T started. Thus DRA prohibits *transaction inversions* (Daudjee and Salem 2006), observed in lazy replication schemes when transaction see stale database states, e.g. a read-only transaction does not observe the writes of the previous transaction performed by the same client. Likewise, using the terminology of (Zhuge, Garcia-Molina et al. 1998), the system provides *completeness*, since non-conflicting transactions are installed in the same order; the one imposed by the acquisition of the *tb_mutex*. This is necessary to preserve the same *reads-from* relations and enable dependability improvement by comparing the corresponding results of SQL operations from different replicas. On the other hand, DRA is more restrictive than a concurrency control mechanism providing SI. DRA synchronises the begin operations, while it is not the case for SI, and as a result the SI's objective of high concurrency (concurrent transactions with disjoint writesets execute successfully without restraints) is not preserved.

When executing in the *optimistic* regime of operation (Figure 3-6), DRA does not generate control read operations because no comparison of the results is performed. Furthermore, a read operation is executed by a replica only if the corresponding result has not been already produced by the other, faster, replica. On the other hand, in both *optimistic* and *pessimistic* regime, DRA offers the same replica control mechanism. The global synchronisation of the transaction boundary operations, begins and commits, and the use of 2PC-DR still apply.

3.1.2. DRA Optimisations

DRA guarantees strict consistency between replicas by imposing the same order of transaction boundary operations (*begins* and *commits*) on both of them. We can optimise the algorithm, when executing in either the *pessimistic* or the *optimistic* regime of operation, in the following ways.

Firstly, we could relax the requirement that the order of *begin* operations is identical on the replicas. As long as no *commit* operation is executed in between a sequence of *begins*, different orderings of *begin* operations are allowed on different replicas. For example, let us consider three concurrent transactions $T_{0,}$ T_1 and T_2 executing over two replicas R_x and R_y. Let us assume a schedule of the transaction boundary operations on R_x is: $c_{0,}$ b_1, b_2, c_1, c_2; then an equivalent order of transaction boundaries: $c_{0,}$ b_2, b_1, c_1, c_2 is allowed on R_y (Figure 3-7). The figure shows that the "granularity" of the synchronisation would change (see the dashed rectangle around the sequence of the two begins): a sequence of two begins, instead of a single one, is executed synchronously on both replicas. In this way a sequence of begin operations would be allowed to execute in parallel, though any commit operation remains executed synchronously and it would be blocked until the sequence of the begin operations is executed on both replicas. Although Figure 3-7 implies execution of transaction boundaries at the same physical time on different replicas, the executions occur with a time lag between them. However it does not invalidate the premise of total order, i.e. equivalent histories of transaction boundaries are preserved.

The tight synchronisation of transaction boundaries can be further relaxed. A sequence, *Commit_SEQ*, of commit operations belonging to non-conflicting transactions (of which the respective writesets are disjoint) can be executed in *different* order on the two replicas. Let us denote the writesets of these transactions, grouped in a set, as *WS*. Similarly to the preceding optimisation of the begin requests,

```
1)  Upon SQL Operation OP from Ti

    A)  if OP is the begin then
        I)      obtain tb_mutex
        II)     begin Ti on both replicas /*create snapshot for Ti*/
        III)    release tb_mutex
        IV)     return to client /*return control*/

    B)  else if OP is a read operation
        I)      execute SKIP procedure
        II)     receive fastest response
        III)    return to client /*return control and the response*/

    C)  else if OP is a write operation
        I)      send the write to both replicas
        II)     if a write-write conflict reported then
            (i)     set transaction abort /*vote for abort*/
        III)    else
            (i)     receive fastest response for the write
            (ii)    return to client /*return control */

    D)  else if OP is an abort /*client sends the abort operation*/
        I)      abort Ti on both replicas

    E)  else /*it is a commit operation - use 2PC-DR*/
        I)      Upon both replicas voted
            (i)     if (an abort vote) /*a write-write conflict reported*/
                (a) abort Ti on both replicas
            (ii)    else /*both ready to commit and no mismatch between the
                        results*/
                (a) obtain tb_mutex
                (b) commit Ti on both replicas
                (c) release tb_mutex

SKIP procedure /*executed by each thread communicating with a replica*/
1)  if no response produced for the read
    A)  send the read to the replica
```

Figure 3-6 Pseudo-code of DRA algorithm when DivRep middleware operates in the *optimistic* regime.

the synchronisation "granularity" would change from a single commit operation to a sequence of commit operations. Ensuring that any begin operation is blocked until *Commit_SEQ* members are executed guarantees the same reads-from relations on both replicas. Although such an ordering of transaction boundaries ensures consistency and allows for correct result comparison, it is too restrictive. To illustrate this point let us introduce a set of readsets, *RS*, that are disjoint with any of the *WS* members. Then replicated database consistency would be preserved if the begin of a transaction having a member of *RS* as a readset is not blocked and executes concurrently with the members of *Commit_SEQ*.

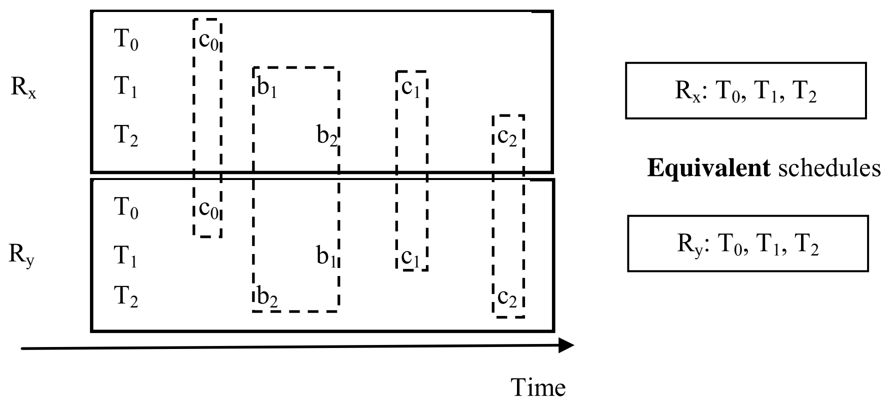

Figure 3-7 Equivalent schedules of transaction boundary operations on two replicas R_x and R_y.

Finally, it seems unnecessary to synchronize the commits of read-only transactions. In that respect there are two possibilities. A commit of a transaction will be synchronized with the transaction boundaries of concurrent transactions if a list of SQL operations, maintained by DivRep, does not contain any write operations or DRA makes sure that the transaction's writeset, WS_i, is empty. The latter is similar to the functionality of other replication schemes (Lin, Kemme et al. 2005), (Kemme and Wu 2005), (Patino-Martinez, Jimenez-Peris et al. 2005) and can be performed using triggers for writeset retrieval or extracting writeset from the transactional logs. These techniques are readily available in many leading database servers, e.g. Oracle, Microsoft SQL, PostgreSQL etc.

3.1.3. Distributed Deadlock Avoidance

To avoid distributed deadlocks, DRA relies on a deadlock prevention scheme that uses a specific parameter, referenced as *NOWAIT*. When the underlying databases guarantee SI, the parameter ensures that an exception is raised as soon as concurrent

transactions attempt to modify the same data item. Figure 3-8 shows the functionality of the parameter using an example when two transactions, T_1 and T_2, execute against a centralized database. Each transaction requires exclusive locks on two resources, A and B, but the order of lock acquisition is different for two transactions. Once T_2 attempts to acquire a conflicting lock, *lock A*, an exception will be raised since T_1 already holds the lock, and T_2 will have to abort. Note that if *NOWAIT* parameter was not enabled the opposite order of lock acquisition would have lead to a deadlock. The behaviour of the parameter is different from *first-committer-wins* and *first-updater-wins* rules (Fekete, O'Neil et al. 2004) since no waiting for one of the transactions to end is necessary. The use of *NOWAIT* might lead to an increase in the number of transaction aborts, and corresponding restarts (Bernstein and Goodman 1981) than if a deadlock detection scheme was used, but incurs no extra overhead needed for the construction of potentially complex *waits-for* graphs, which incur the principal overhead in deadlock detection schemes. The use of the *NOWAIT* parameter seems to be attractive due to its simplicity, especially for the workloads with low probability of deadlocks/aborts.

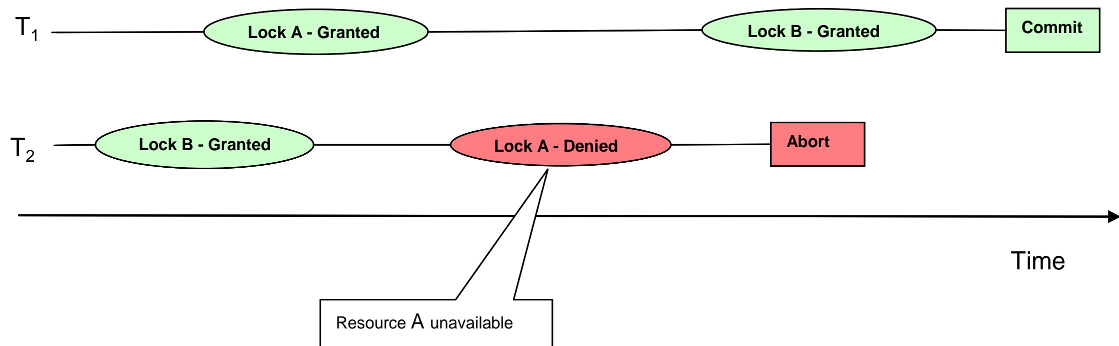

Figure 3-8 Deadlock avoidance using *NOWAIT* parameter on a non-replicated database exemplified with two concurrent transactions, T_1 and T_2 (the corresponding *begin* operations are omitted). As soon as transaction T_2 requests a conflicting lock for resource A, the database server will raise an exception and T_2 will have to abort.

A number of database servers offer the behaviour of *NOWAIT* parameter, such as Oracle, PostgreSQL, InterBase, Firebird etc., although the respective implementations might differ. For example, on Firebird the *NOWAIT* parameter is specified for a database connection, while on PostgreSQL the behaviour is available through the SELECT … FOR UPDATE operation, which locks the selected rows against concurrent updates.

In order to avoid distributed deadlocks in a replicated database not all, but *exactly n-1* replicas should be configured with the *NOWAIT* parameter (where *n* is the number of

replicas). Otherwise, if the number of replicas that enabled *NOWAIT* parameter is less than *n-1*, a distributed deadlock, that spans replicas which have not enabled the parameter, could be observed. Hence the liveness in the replicated database will be compromised. We assume deployment of two replicas with DivRep and one has the *NOWAIT* parameter enabled. Despite enabling *NOWAIT* parameter on one of them, other types of concurrency conflicts might be reported, since the replicas use 2-Phase locking for write operations. In particular it is possible that a replica, which has not enabled *NOWAIT*, reports a centralized deadlock (when concurrent transactions executing on the same replica try to acquire a set of locks in different order). In this case DivRep follows the decision made by the replica and aborts the *victim* transaction. Similarly it should be noted that on the replica which *has* enabled *NOWAIT* an exception due to the *first-updater-wins* rule could be observed, i.e. not all exceptions would be raised as a result of the *NOWAIT* parameter.

One is interested in how much impact *NOWAIT* parameter has on the abort rate. We reason about the matter informally using Figure 3-9. Let us assume an FT-node, a system with two replicas, in which only one of the replicas, R_A, has *NOWAIT* parameter enabled (clearly FT-node is a special case of a system with more than two replicas). Two overlapping transactions, T_i and T_j, both try to modify data item *x*. The moment transaction T_j tries to acquire exclusive lock on *x*, *NOWAIT* exception will be raised and the middleware will abort the transaction on both replicas. Following the abort of T_j on R_B the first-committer-wins rule is enforced, exclusive lock will be granted to T_i and the transaction will successfully commit on both replicas.

However had *NOWAIT* parameter been enabled on replica R_B too, transaction T_i would have been also aborted, following the *NOWAIT* exception raised as part of T_j execution. Hence, in the cases when more than one replica has *NOWAIT* parameter enabled the deadlock avoidance scheme might result in *unnecessarily* many aborts. This is not possible in DivRep, however, where only *one* replica enables the parameter (in this way DivRep obeys that n-1 replicas have *NOWAIT* enabled). Moreover, despite having *NOWAIT* on both replicas, *unnecessary* aborts might be an infrequent event in practice: it is likely that the abort of T_j happens earlier, in global calendar time, than the request for the writing of *x* by T_i on R_B – in this way T_j would release the locks, *NOWAIT* exception would not be triggered by R_B and T_i would commit successfully.

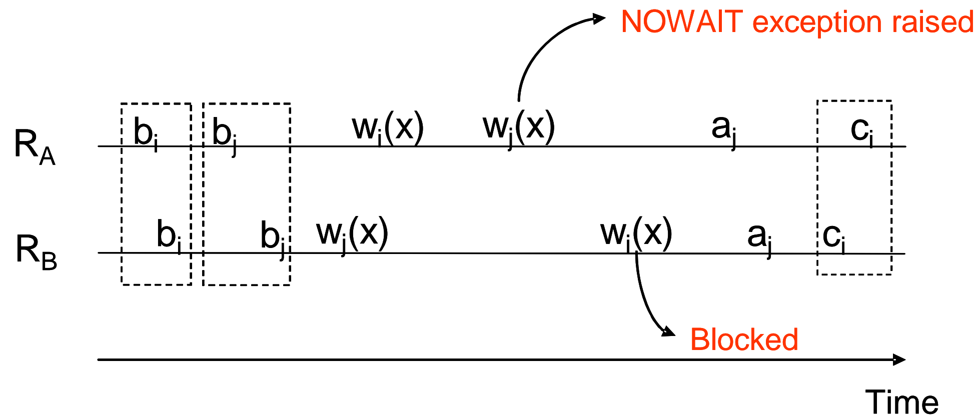

Figure 3-9 Deadlock avoidance using *NOWAIT* parameter in a replicated database system with two replicas, R_A and R_B. Two transactions T_i and T_j execute concurrently and access the same data item x. Only transaction boundary operations (begins (b), aborts (a) and commits (c)) and the write of the common data item are shown. The interaction between the replicas and the middleware and similarly between the clients and the middleware is omitted for clarity.

There is an exception to the claim that DivRep does not produce *unnecessary* aborts: the aborts are possible in specific situations where one of the database engines in an FT-node provides deadlock detection mechanism, which might make contrary decisions to a replica with *NOWAIT* enabled. Consequently, it is possible that two replicas resolve conflicting transactions in a different order. An instance of this situation is depicted in Figure 3-10, which shows executions of two transactions T_i and T_j on two replicas R_A and R_B. Replica R_A will report concurrency conflict exception as soon as T_j tries to acquire lock for data item *x* and as a result the transaction will have to be aborted. Correspondingly, on replica R_B (which has *NOWAIT* disabled) a deadlock detection scheme will decide that T_i should be aborted due to the centralised deadlock. Without imposing execution determinism in DivRep both transactions will be aborted. However, the particular series of events are unlikely - the value of the deadlock timeout should, e.g. according to the best practice guides for database administrators, exceed the typical transaction time, and in that way avoid possibility that inconsistent decisions are made by different replicas (transaction T_j would be aborted before deadlock detection mechanism is triggered on R_B and as a consequence transaction T_i would commit).

FT-node uses two database servers, one of which has *NOWAIT* parameter enabled. When selecting the pair of servers a particular database engine might not offer *NOWAIT* parameter, and ruled out as an inappropriate choice for FT-node. Nonetheless the selection process should verify if the functionality of *NOWAIT* could be simulated using other configuration parameters – many database servers offer a

lock timeout parameter that prevents indefinitely long blocking, e.g. *LOCK_TIMEOUT* in Microsoft SQL server or *innodb_lock_wait_timeout* in MySQL. Commonly, setting the values of the parameters to zero simulates *NOWAIT* functionality; or at least fine-grained timeout values (e.g. in milliseconds) could achieve the same.

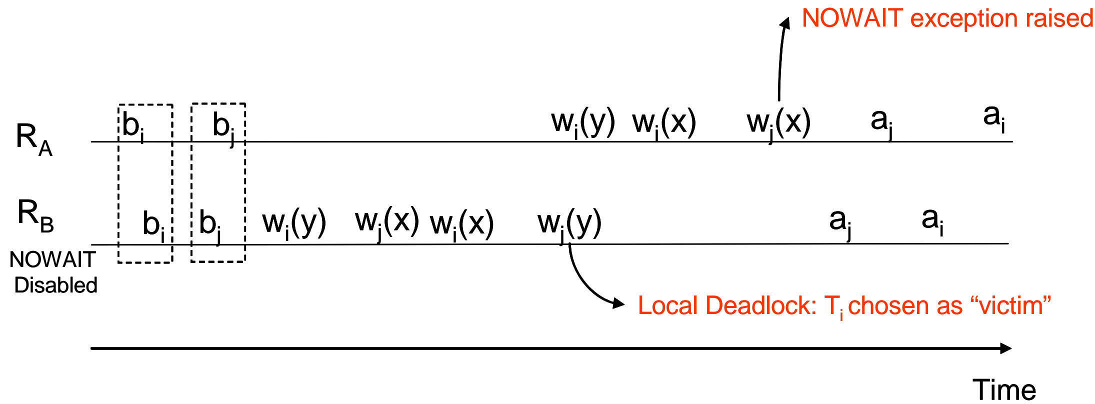

Figure 3-10 An example of *unnecessary* aborts in DivRep. Two transactions T_i and T_j execute concurrently and both access the two data items, *x* and *y*, on the two replicas, R_A and R_B. Only transaction boundary operations (begins (b), aborts (a) and commits (c)) and the writes of the common data items are shown. The interaction between the replicas and the middleware and similarly between the clients and the middleware is omitted for clarity.

3.2. Correctness of DRA

3.2.1. Safety

In this section we provide the proof of correctness of DRA algorithm. There are two versions of the proof. The first one is based on the definition of *1-copy-SI* (Section 2.3.2) and the second one uses the formalism of CSI (Section 2.3.2). The correctness of DRA is not limited by the use of two replicas with DivRep – it is ensured when an arbitrary number of replicas are deployed.

Safety of DRA using 1-copy-SI

The following definition, *Strict 1-copy-SI*, is based on the corresponding definition of 1-copy snapshot isolation from (Lin, Kemme et al. 2005). We use the definition of SI-Equivalence (Section 2.3.2) for formalising *Strict 1-copy-SI.* The difference is in removing the ROWA restriction, where reads are executed only at a local site. We are interested only in the set of committed transactions (Bernstein, Hadzilacos et al. 1987).

Strict 1-copy-SI. *Let S be a set of schedules, R a set of replicas and T a set of submitted transactions. Let S^k be an SI-schedule over the set of committed transactions on replica R^k. We say that R provides Strict 1-copy-SI if all schedules from S are SI-equivalent.*

Assumption 1: The underlying replicas provide Snapshot Isolation.

Proposition 1: All replicas commit the same set of transactions.

Proof: After a transaction is submitted, it commits either on all replicas or at none. This follows from the fact that a transaction termination is performed using an atomic commitment protocol, 2PC-DR (see the part E. in Figure 3-5), where all replicas agree on an outcome, commit or abort, i.e. uniform agreement is guaranteed.

Q.E.D.

Theorem 1. DRA guarantees Strict 1-copy-SI.

Proof: Assume it does not. Then there exists a pair of schedules $(S^1, S^2) \in S$ and a pair of transactions $(T_i, T_j) \in T$ for which the following holds:

i. $WS_i \cap WS_j \neq \{\} : (c_i < c_j) \in S^1$ ***and*** $(c_j < c_i) \in S^2$.

or

ii. $WS_i \cap RS_j \neq \{\} : (c_i < b_j) \in S^1$ ***and*** $(b_j < c_i) \in S^2$.

Both *i.* and *ii.* are impossible because *Proposition 1.* holds and the transaction boundary operations are executed atomically in DRA – the same order of transaction boundary operations is preserved on the replicas (see 1.E.I.ii.(a-c) and 1.A.(I-III) in Figure 3-5).

Q.E.D.

The above proof holds for both *pessimistic* and *optimistic* regime of DRA. In the *optimistic* response regime, some of the reads might be *skipped*, but the uniform agreement and the identical order of transaction boundary operations is preserved.

Safety of DRA using CSI

Assumption 1: Underlying replicas ensure CSI

Theorem 1: DRA ensures CSI

- DRA ensures conditions C1.1 and C1.2 of CSI (see 2.3.2) because the underlying replicas are assumed to guarantee CSI where only updates of committed transactions are visible i.e. no *dirty reads* are allowed.

- DRA guarantees condition C1.3 of CSI - every transaction observes the *last* committed snapshot on any replica.

  Assume C1.3 was not ensured. Then it is possible for a transaction to read an "old" snapshot (we denote this property $\neg$C1.3):

  $\exists T_i, T_k, X_j$ such that $R_i(X_j) \in h$ and $W_k(X_k), C_k \in h$:

  commit(T_j) < *commit*(T_k) **and** *commit*(T_k) **<** *start*(T_i);

  - $\neg$C1.3 is possible **only** if a replica produces such a schedule since every transaction starts atomically on both (all) replicas using *tb_mutex* and an identical order of begins and commits is ensured (see 1.A.I-III and 1.E.I.ii.a-c in Figure 3-5).
  - However $\neg$C1.3 contradicts *Assumption 1*.

- DRA enforces C2 (CSI Commit Rule)

  Assume it does not. Then it is true that impacting transactions are allowed (we denote the property $\neg$C2):

  $\exists T_i, T_j$ such that $C_i, C_j \in h$:

  start(T_i) < *commit*(T_j) < *commit*(T_i) **and** *writeset*(T_i) $\cap$ *writeset*(T_j) $\neq$ {}

  This is impossible since replicas provide snapshot isolation and a transaction will be aborted by DRA if an "impact" i.e. *write-write* conflict, has been detected on any of the replicas (see 1.C.II.i. in Figure 3-5).

Q.E.D.

3.2.2. Liveness

In addition to the guarantee that no distributed deadlock is possible, i.e. that Liveness 1 (Section 2.2.4) is guaranteed, DRA ensures a higher degree of liveness. Since it is assumed that replicas guarantee snapshot isolation in DivRep Liveness 2 is guaranteed.

As to prevent repeated aborts and guarantee progress of a modifying transaction, DRA could use the following technique. After a transaction T_i had been aborted a predefined number of times DivRep enters a special mode of operation, *write bottleneck* (Popov, Strigini et al. 2004), where no concurrency of the modifying transactions is allowed – only one modifying transaction executes at the time. Once T_i has been executed successfully the middleware restores the default regime of operation. In this way DRA would guarantee that any transaction, read-only or modifying, *eventually* commits. Note that this property is not as strict as Liveness 3

because the predefined number of aborts might precede a successful transaction termination.

3.3. Hybrid Approach of DivRep

DivRep middleware is configurable to run in different regimes of operation depending on the specific client requirements. The *pessimistic* regime of operation offers improved fault–tolerance, by comparing the results of SQL operations from different replicas, while the complementary *optimistic* regime delivers performance improvements by executing the read operations only on a single replica.

It is worth noting that these two regimes are not mutually exclusive and they can be combined into a *configurable quality of service*. By deploying *learning capabilities*, e.g. using Bayesian inference (Gorbenko, Kharchenko et al. 2005), the middleware may process the individual SQL operations switching intelligently between different regimes of operation. The switch between the regimes will be driven by *confidence* gradually built by the middleware that a particular type of operation is unlikely to cause a disagreement between the responses of the deployed diverse replicas. Before the predefined level of confidence is reached, whenever the middleware recognises the operation it will process it under the *pessimistic* regime. As the number of instances of the same type of operation (e.g. the same query but with different values of the parameters) grows, and no disagreements are observed between the responses of the replicas, so will the confidence that the particular type of operation is unlikely to lead to disagreements between the diverse replicas. Eventually, the predefined level of confidence will be reached, from which point the middleware will execute the subsequent instances of the same operation under the *optimistic* regime. A disagreement between the replica responses during the learning period will either lead to the middleware processing all future instances of the operation under the *pessimistic* regime or will require a significantly increased number of identical responses for the predefined level of confidence to be reached.

In the cases where multiple applications execute against the same FT-node, and some of them are not subjected to runtime assessment using learning capabilities, a special care has to be taken. This is because database inconsistencies might be introduced with the applications that do not use the hybrid approach for dependability assurance, and determining the switching point between the two regimes of operation, for

applications that seek improved dependability, could be invalidated. To guard against these possibilities DivRep can initiate periodic consistency checks, after the switch between the regimes had occurred and executing in the *optimistic* regime of operation is taking place. This will also help reveal the cases where, despite the initial execution in the *pessimistic* regime of operation, an inconsistency is triggered after the switching point. At the same time dependability improvement might be sought by augmenting the *optimistic* regime of operation with a fault-tolerant feature – inconsistencies in a replicated database can be uncovered by executing control-reads as part of the *optimistic* regime.

In parallel with determining the switching point between the two regimes of operation learning capabilities could be used for determining which replica is faster for a particular type of read operation. Consequently, as a part of a new regime of operation, the read is executed only on the faster replica. The load of the read operations would be divided between the replicas once the middleware *learns* which replica is the fastest for all the reads. This could be more efficient than the *skip* feature because no read operation would be executed on more than one replica. Some feedback data would need to be provided to the load balancing technique so that changes in response times, e.g. a faster server starts to work more slowly under different workload, are detected. The technique would resemble ROWA, but it would be more flexible than the well-known approach, because the middleware would have alternatives in deciding on which replica to execute a particular read operation e.g. using additional load balancing information a read could execute on a slower replica in the cases where it is being subjected to a lighter load.

3.4. Discussion

This chapter has introduced a middleware-based database replication solution. We have described different modes of operation of DivRep middleware, explained replica-control algorithm and stated the possibilities for a hybrid solution. Also, we have proved the correctness of DRA algorithm ensuring replica consistency. What follows is the discussion about strengths and weaknesses of DivRep, and possible changes for improved dependability and better performance.

3.4.1. Comparing DivRep to Other Replication Techniques

A standard fault-tolerant architecture (Figure 3-1) dictates adjudication of the responses from diverse replicas. The adjudication is applied at the level of individual operations. Hence, fault-tolerance will lead to performance penalty and the FT-node is guaranteed to perform worse than the slower replica. In addition the transaction termination is achieved using an implementation of a 2PC protocol. Clearly these characteristics limit the scalability of the replication protocol, which, however, is not a serious problem since we envisage FT-node as primarily consisting of two diverse replicas. The schemes adopted for practical database replication provide no protection against design faults. A common assumption is made that node crashes are the main concern, an assumption under which various optimistic regimes of operations are used, e.g. ROWA. These do not require operation adjudication and as a result the adjudication overhead is simply avoided. Thus, there is no scope for trading-off intelligently performance for dependability assurance.

There are numerous applications which use operations that are handled correctly by the deployed DBMSs. Even if diverse DBMSs are deployed most of the operations will be handled correctly by the diverse replicas. Thus, most of the time adjudicating the responses of diverse replicas will reveal no discrepancy, making the adjudication overhead appear as a waste of time. The point, of course, is that we will never know which operation will turn out to trigger a fault in the DBMSs and revealing a discrepancy between the replica responses. In some extreme cases, however, *one may know with certainty*, that all the operations used by the application will be processed by the DBMSs correctly; hence one may be prepared to use regimes in which the adjudication is eliminated. One such example is the implementation of the *optimistic* regime. Its advantage compared with the well-known ROWA regime of operation lies in the fact that under ROWA the load is statically distributed between the replicas – in the ideal case a fair load-balancing between the replicas is sought. Instead, when the FT-node operates under the *optimistic* regime its diverse replicas *naturally get the load that they are better at executing*. As a result the *optimistic* mode has the potential for good performance.

DivRep is a type of update everywhere replication in which the role of the *delegate* server, a node to which client requests are submitted (Weismann, Pedone et al. 2000), is performed by the middleware. Although the possibility of executing SQL

operations on any of the replicas, instead of using a primary copy approach, seems appealing, there are reasons why the traditional update everywhere approach is not always called upon for performance improvement. The most important one is that the performance is dependant on the workload. If there is a significant number of update operations in the workload the processing will be replicated on all sites. This is exacerbated in DivRep where extra dependability assurance, guaranteed by the *pessimistic* regime, necessitates the execution of read operations on all replicas. Furthermore the *linearity*, which denotes propagation of operations to all replicas, in message exchanges (Weismann, Pedone et al. 2000) hinders the performance of DivRep.

DivRep uses a simple approach to replica control: execution of transaction boundary operations are controlled using *tb_mutex* (Figure 3-5) to ensure replica consistency and the conflict detection is performed using the concurrency mechanism of the underlying servers. In this way no complex replica control mechanism is performed in the middleware as is the case in many other solutions (Plattner and Alonso 2004), (Patino-Martinez, Jimenez-Peris et al. 2005), (Lin, Kemme et al. 2005), (Pedone and Frolund 2005). Conflicts are detected early in DivRep, as soon as a replica reports them, instead of waiting for a certification phase to complete and report them. Moreover, DivRep is advantageous in comparison to the replication schemes where declaration of transaction characteristics prior to their execution is required, e.g. *pre-declaration* of the tables and the particular types of operations (read or write) used in a transaction (Amza, Cox et al. 2003). DivRep has an advantage over other database replication solutions in which SI is assumed to be guaranteed by the replicas, such as (Kemme and Wu 2005), (Daudjee and Salem 2004), (Elnikety, Zwaenepoel et al. 2005): it guarantees SI as found on centralised database systems, where each transaction operates on the *latest* database snapshot i.e. it provides Conventional Snapshot Isolation as described in (Elnikety, Zwaenepoel et al. 2005). Thus if a transaction T_2 starts after a committed transaction T_1, it is guaranteed that T_2 observes the committed database state that includes T_1's changes. Note that according to the original definition of SI in (Berenson, Bernstein et al. 1995) it would be possible that system chooses an older snapshot, excluding T_1's changes, for T_2 to operate on, despite the fact that T_2 starts after T_1 commits. In this way DivRep provides read-only transactions with the most recent snapshot, a property commonly unavailable in other replication solutions, which permit stale data to be read. In centralised databases, by

providing the latest snapshot the abort rate of update transactions is reduced, as the number of overlapping transactions is reduced.

3.4.2. Possible Changes to DivRep

Error detection of the *pessimistic* regime requires consistent snapshots of data from both replicas. One might be interested in what would be the consequences if the replication protocol of DivRep was modified so that instead of the *latest* committed snapshot a transaction observes an "older" one. In this way DivRep would ensure GSI (Elnikety, Zwaenepoel et al. 2005). The *pessimistic* regime would still use the comparator function for improved error detection but it would not (necessarily) operate on the snapshot installed by the last committed transaction. It is, however, far from obvious that this could bring any performance benefits. This argument is workload-dependant and if a high conflict rate was observed this change to DivRep would have negative impact on performance, since the probability of overlaps between transactions would be higher. For example, let the following schedule of transaction boundaries, belonging to two transactions T_1 and T_2, be produced: b_1, b_2, c_2, c_1. Then the logical start of another transaction, T_3, which conflicts with T_1 will be crucial for decreasing the abort rate: if b_3 is placed immediately after c_1 (the latest snapshot is available) then the conflict will be avoided; but if it is placed in between c_2 and c_1 (the changes of c_1 are unavailable) then T_1 and T_3 will overlap and the conflict will lead to an abort.

It is possible to extend fault tolerance features of DivRep by devising an error detection mechanism for handling SQL DDL (Data Definition Language) operations, which are used to define database structure (e.g. *CREATE TABLE* operation), and stored procedures (precompiled pieces of code available to applications accessing database through DBMS APIs). Although DDL operations are usually less frequent than DML operations, ensuring the consistency of their results on different replicas is as important. Nevertheless, this is far from a trivial task since comparing the results of DDL operations from diverse replicas requires the access to different database metadata information. Concerning stored procedures, the task is more intricate. Let us assume that comparing the effects of a stored procedure execution on diverse replicas could be done using the returned results. However this is not adequate. Firstly, there are cases when a stored procedure does not return a result. Secondly, the execution of a stored procedure could involve both DMLs and DDLs and, thus, developing a

generic solution for checking the consistency of the results created by a stored procedure across diverse replicas is not obvious. Database *triggers*, pieces of code that automatically execute in response to an event, could help in providing error detection among results of DDL operations and stored procedures. In particular, *schema-level triggers*, which exist in the Oracle DBMS and fire when a database schema object is modified, could be useful.

We propose the use of the FT-node for tolerating design faults in order to increase failure detection. It is evident, however, that the middleware itself represents a single point of failure. Standard techniques, such as primary-backup replication (Budhiraja, Marzullo et al. 1993) or implementing decentralised DivRep, could be used to alleviate this problem and improve availability and scalability. The middleware is likely to be relatively simple and, thus, we can achieve high confidence in its being implemented correctly, i.e. free of design faults. Therefore, presuming fail-stop (only crashes) failures becomes reasonable assumption. Hence the solutions based on this assumption become relevant.

In order to enhance DivRep so that the replicas are able do decide on the outcome of a transaction (to commit or abort) even in the presence of failures, it is possible to substitute *2PC-DR* (Figure 3-5) protocol with an implementation of the Non-Blocking Atomic Commitment (NB-AC) protocol. For example the well-known Three-Phase Commit algorithm (Skeen 1981), which assumes synchronous systems and bounded communication delays, can be used to solve NB-AC problem. Alternatively DivRep could be equipped with Paxos Commit algorithm (Gray and Lamport 2006) in order to solve the atomic commitment problem between the replicas and the comparator function. This is likely to decrease response time at the expense of complicating the replication protocol.

One possibility to further improve the performance of the *optimistic* regime of DivRep is to introduce *cancellation* of read operations. Load balancing using *skip* feature is effective only in certain scenarios. Let us assume there are two replicas, R_x and R_y, executing a read operation *r(a)*, as part of transaction T_i. If the replica R_x starts and completes *r(a)* while the other replica, R_y, is executing the preceding operations in T_i, the *skip* feature will cause replica R_y to leave out the execution of *r(a)*. However the skipping is impossible if the executions of *r(a)* overlap, in global calendar time, on two replicas (it is in fact more restrictive: no skip occurs if DivRep receives the result to *r(a)* only after the thread serving the slower server has sent the

read to the replica). The best that the middleware can do in that situation is to cancel the execution of the read on the slower replica, once it obtains the result of *r(a)* from the faster one. Nevertheless, the cancellation would carry a performance overhead and it is unclear whether cancellation will improve the situation or make things worse. The effectiveness would depend on its overhead and the decreased load on the slower replica. From implementation point of view the cancellation requires the support from the database engine, a feature not available on all servers, and a separate thread of execution on the client side due to which undesirable *race conditions* might ensue. Moreover this could lead to a wrong SQL operation being cancelled. For example, in between issuing a cancel operation, from a dedicated client thread, and executing it in the database, the long-running read (the operation to which the cancel was directed) might finish and another operation would start executing. Therefore, the cancel will wrongly terminate the execution of the subsequent operation.

DivRep uses active replication with the aim to compare results and provide error detection. One might be interested in using an alternative for the active replication in order to improve performance. To that end *deferred writes* technique (Bernstein, Hadzilacos et al.) (Section 2.3.1) is one such possibility. For example, aborting a transaction during its execution on a local replica, before the updates are sent to the other one, would be less costly. Likewise, by localising the execution of multiple updates on a replica and propagating them together, the number of messages in the network would be reduced. However, the use of deferred writes is unacceptable for DivRep, at least when the *pessimistic* regime of operation is considered. The technique involves execution of the writes on a local replica and propagation of the respective results (e.g. the redo log records) to the other replicas. This would prevent the error detection deployed in the *pessimistic* regime because of the following:

- The input to the comparison function (see Figure 3-5) is indeterminate – the changes produced by the local replica are just applied to the remote one.
- Incorrect results, produced by a faulty replica, would be propagated.

This could be alleviated with the propagation of full SQL operations, instead of the log records, to the remote replica. Even in this case, the advantage of executing result comparison in parallel with SQL operations will be lost - the results would have to be compared in the critical path, in the end of the transaction. Moreover, the active replication will have to be continued for SELECT operations, so that the results of the reads could be compared.

4. Experimental Evaluation of DivRep Performance

There are three principal means of acquiring knowledge:
observation of nature, reflection, and experimentation.
Observation collects facts; reflection combines them;
experimentation verifies the result of that combination.
Denis Diderot

This chapter provides an extensive evaluation of the performance when diverse database servers are used for replication. In the course of the evaluation we have performed a multitude of experiments, varying different experimental parameters such as transactional mix, load, FT-node and server configurations etc. The different types of experiments we have performed can be coarsely categorised as follows:

- Initially, to justify motivations for the use of diverse DBMSs for database replication, we have performed experiments without the use of DivRep middleware. These experiments focused on the exploration of variability in performance of diverse servers.
- Then we conducted a performance comparison between DivRep middleware employing diverse servers and DivRep middleware employing non-diverse products. This comparison is based on a specific mix of transactions where multiple clients were executing a read-only mix of transactions in parallel with a single modifying client communicating with the replicated system. Like in the initial set of experiments, only an early prototype of DivRep middleware was employed, without the use of DRA algorithm to ensure the consistency among deployed replicas.
- Subsequently we present the results of DivRep scheme where multiple modifying clients were communicating with the replicated system. The central goal of these experiments was to evaluate the performance implications of increased dependability guaranteed with DivRep by comparing it to a known database

replication solution based on ROWA approach. A fully operational DivRep middleware was used in this set of experiments.

We have performed a thorough analysis, beyond the use of summary statistics such as mean, variance etc., of the experimental results in order to get a more informative insight of DivRep's performance. We show that in certain cases, when users are primarily interested in improved performance and possible dependability deterioration can be tolerated, DivRep scheme can deliver better service than the one provided by the conventional non-diverse replication.

Moreover, we have devised a possible solution for decreasing the performance overhead incurred by sequential execution of transaction boundary operations in DivRep. The solution is based on the use of process prioritisation for database servers processes. We have performed a set of experiments to evaluate the effectiveness of the solution.

4.1. Test Harness

One of the contributions of the thesis is the development of a test harness for performance evaluation of different database replication solutions as well as non-replicated server configurations. The testbed (Figure 4-1) provides for rigorous and replicable experimentation with various database servers' configurations. It is implemented as a distributed application using Java programming language in a form of a three-tier architecture. The client and middleware parts of the testbed execute as a multithreaded application. They include the following:

- *A client application* (*TPC-C.jar* in Figure 4-1). This is our own implementation of TPC-C (TPC 2002a), an industry standard benchmark for On-line Transaction Processing (OLTP) (see Section 2.4). A number of TPC-C parameters are configurable in the testbed, e.g. the probability of execution of each type of transaction. A separate thread executes the logic of each TPC-C client. A dedicated database connection, opened in the beginning of an experiment and closed upon its end, is used by each TPC-C client.
- *A replication middleware* (*Replication Middleware.jar* in Figure 4-1). The middleware is based on the multi-master (Wikipedia 2007) synchronous replication approach. Multi-master systems allow writes to be submitted at multiple replicas independently and exchange them synchronously (before transaction ends) or

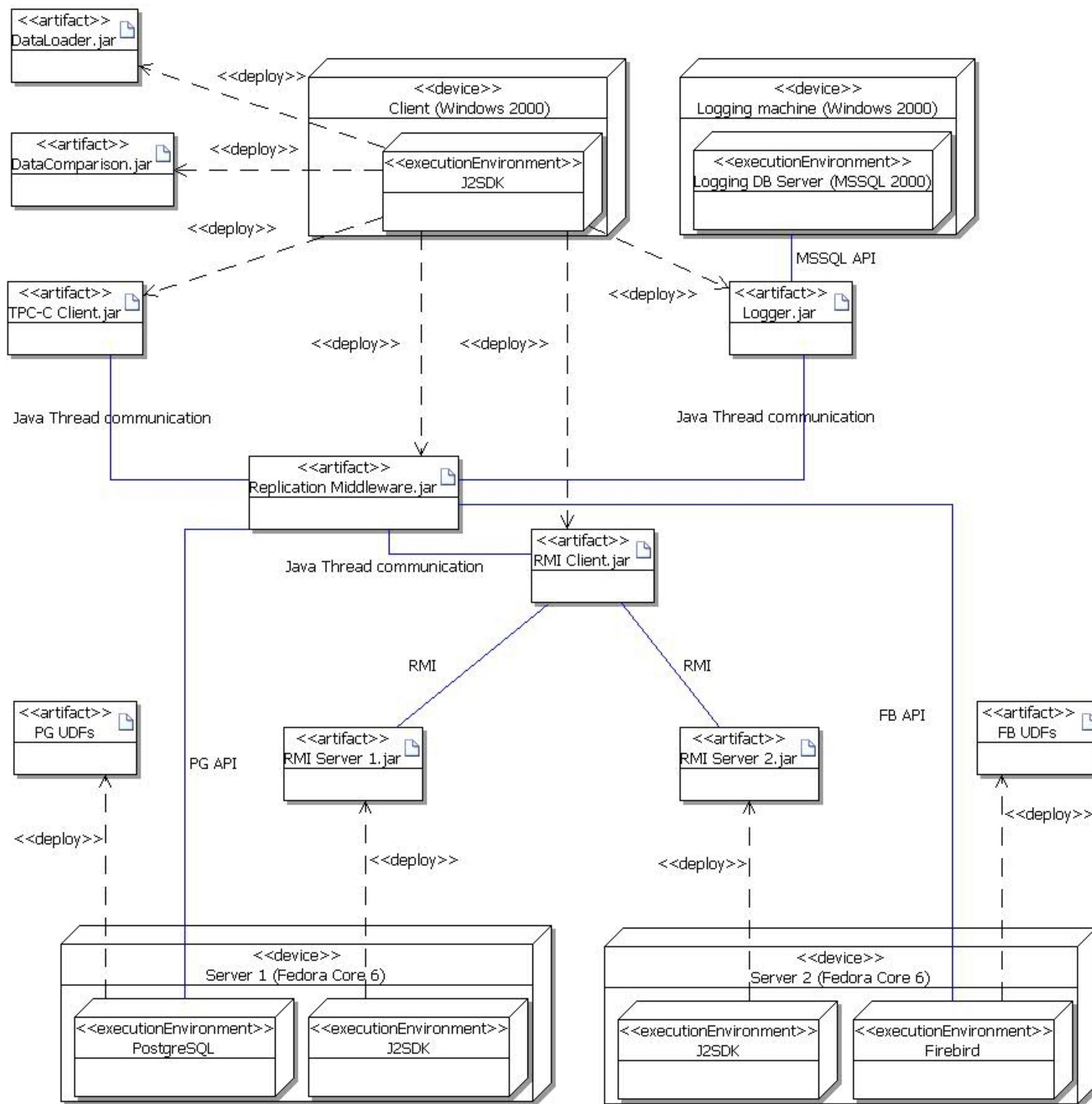

Figure 4-1 The architecture of the testbed for the database replication as a UML 2.0 deployment diagram.

asynchronously (after transaction ends). The replication is done at the SQL operation level (i.e. the applications are assumed to access the data via SQL operations). The middleware offers to the clients a JDBC interface for database access. For each client the middleware spawns two threads, one per each replica in the FT-node (Figure 3-1). The middleware provides for experimentation with various database replication schemes and allows for different combinations of servers to be deployed. The servers can belong to the same database vendor (*non-diverse* redundancy) or servers from different database vendors could be used (*diverse* redundancy). Hence, comparison of the performance of our diverse replication scheme (DivRep) against a known non-diverse alternatives, e.g. read ROWA schemes, is possible.

- *Logging utility* (*Logger.jar* in Figure 4-1) The utility collects experiment's measurements and stores them in a *log* database on a remote machine. Inter-thread communication is used between the replication middleware and the logging utility. The log database holds extensive information about each experiment e.g. details about all transactions, SQL operations, exceptions raised, as well as details of possible inconsistencies between the results from different replicas. Appendix A contains the schema of the log database.
- *Database comparison utility* (*DBComparison.jar* in Figure 4-1). The utility compares the states of databases after experiments and reports possible inconsistencies. It reads data items from the database servers one-by-one and reports an inconsistency if corresponding values do not match. When an experiment involves more than one modifying client we cannot compare the states of databases across experiments, due to the inherent non-determinism in the client application. In those cases we, only, compare the states of the replicas at the end of the experiment in which redundant server configurations were used.
- *Data loader utility* (*DataLoader.jar* in Figure 4-1). The utility creates and populates TPC-C databases on (diverse) servers following the specification detailed in the standard. Configurable parameters of *Data Loader* enable us to create differently scaled databases.

Replication middleware.jar implements the *Comparator function* (Section 3.1.1). The control reads for the three types of write operations (DELETE, INSERT and UPDATE) are generated as follows:

- DELETE FROM *table* [WHERE *condition*]

We first extract the value from the FROM clause to obtain the table name (notice that the possible extension to the SQL standard (ANSI 1992) which allows for joining of the tables in the FROM clause is catered for in this case). If the write operation has no WHERE clause the control read has the following syntax: SELECT * FROM *table*. Otherwise we extract the value of the WHERE clause and produce the corresponding control-read: SELECT * FROM *table* WHERE *condition*.

- INSERT INTO *table*
  [(*column* [, ...])]
  { VALUES ({ NULL | *expression* } [, ...]) | *query* }
  When the column list is specified, the first part of the control-read operation has the following syntax: SELECT *column [, ...]* FROM *table*. The second part, i.e. the WHERE clause of the control-read, is built by pairing each value of the column list with the corresponding value from the VALUE clause, or the supplied *query*, in the original INSERT operation. As long as the primary key column(s) are specified in the columns list, only the rows inserted with the INSERT operation are retrieved. However, it is possible that the table has a sequence generator (ANSI 1992) as the primary key and that the column list does not contain it. Therefore, multiple rows, with the same values of the specified columns, could be retrieved by the control-read. This is unlikely, however, since in these cases usually the set of specified column values in the original INSERT operation, without the omitted sequence generator value, uniquely identifies a row in the database.
  If the column list is not specified the values of all the columns must be supplied by the original INSERT operation. This is the standard behaviour of INSERT operation as specified by SQL standards. We do not cater for extensions of the SQL functionality by some database engines, whereby default values are inserted when column list is omitted but not all the table's columns are filled from the VALUES clause or the *query*. We read the database metadata to retrieve the list of the column names in the *table* (we do this only once, in the beginning of an experiment, for each table and store it for future use). Subsequently we build the SELECT clause of the control-read operation using these column names. Analogous to the case when the column list is given, the WHERE clause of the control-read operation is generated by pairing the column names with the corresponding values specified in the VALUES clause, or the *query*, of the original INSERT operation.

One observation is worth pointing out. If the table does not have a primary key constraint, and the supplied values of the INSERT operation match existing row(s), the control-read returns excessive result, i.e. additional rows, with the same values as the ones inserted, would be retrieved. However, these circumstances are rare since database normalisation, a technique for designing relational databases without data anomalies (different structural and logical problems), requires the use of primary keys (Date 1994).

- UPDATE *table*
  SET { *column* = { NULL | *expression*} [, ...]
  WHERE *condition*
  The SELECT clause of the control-read is generated by extracting the column names from the SET clause of the UPDATE operation. The FROM clause of the control-read is generated using the *table* name. The arbitrary WHERE clause is the same in the control-read and the original UPDATE operation – this is because we require the condition in WHERE clause to include the primary key of the updated table. In this way, only the rows affected by the UPDATE are retrieved by the control-read operation.
  We, additionally, relax the requirement that the primary key has to be specified in the WHERE clause. This, however, comes with an expense in a general case – it is possible that additional rows apart from the modified ones are retrieved by the control-read. Also, even after the requirement is relaxed, if there are overlaps of the columns in the WHERE clause with the columns in the SET clause a special care has to be taken. In the case of the overlaps, values of the columns specified in the SET clause should take precedence, i.e. they should be the ones used in the control-read operation. Let us look at an example. If the original write operation is UPDATE t1 SET x = 100 WHERE x = 0, the control-read is SELECT x FROM t1 WHERE x = 100. It is possible, however, that if the column x is not the primary key (generally primary keys do not change and some DBMSs even dictate that by preventing modification of primary keys through an UPDATE operation) there had been rows with x = 100 value in the database before the update took place. Hence the control-read would retrieve more rows than modified by the write operation.

In addition to the database servers themselves responsible for handling the client transactions, the server part of the testbed (the third tier) includes a Remote Method Invocation (RMI) server run as a daemon (a background process) on each database

server machine (*RMI Server 1.jar* and *RMI Server 2.jar* in Figure 4-1). The main purpose of the RMI server is to enable maintenance tasks. For example, database backups, restores and machine reboots between experiments are performed as needed to ensure the identical initial state of system resources and databases used in the experiments. To perform these tasks each of the RMI servers communicates with the client machine through the RMI client (*RMI Client.jar* in Figure 4-1). Additionally, we developed User Defined Functions (*PG UDFs* and *FB UDFs* in Figure 4-1) as part of the mechanism for minimising replication overhead (a detailed description is given in Section 4.5).

The testbed consists of four physical machines:

- *A client machine* (*Client (Windows 2000)* in Figure 4-1). The TPC-C Client application and the middleware execute on this machine although the clients and the middleware can be deployed on different machines. We have monitored the resource utilisation on the client machine before performing measured experiments. The monitoring showed that the client machine does not become a bottleneck in any of the experiments.
- *A logging machine* (*Logging Machine (Windows 2000)* in Figure 4-1). The experiment results are stored on this machine. The data is stored using a Microsoft SQL server. The selection of Microsoft SQL server is mainly for convenience in analysing the results due to the familiarity of the author with Transact-SQL (Microsoft 2000). The logging itself does not depend on any Microsoft SQL proprietary feature and the database server can be replaced by a different product.
- *Two server machines* (*Server 1 (Fedora Core 6)* and *Server 2 (Fedora Core 6)* in Figure 4-1). These machines host the database servers under the experiment, which are offered by either the same or different vendors. In our studies we have mainly used two open-source database servers, namely Firebird and PostgreSQL (*Firebird* and *PostgreSQL* in Figure 4-1). These two database servers are denoted as FB and PG, respectively, in the rest of the document. Different configurations of servers can be used: e.g. a single server – 1FB, 1PG, or two servers – 2FBs, 2PGs, 1FB1PG. In our initial studies we have used Firebird's predecessor – Interbase (IB). Borland (Borland 2007) offered IB as an open source product while in version 6. However the company reverted to the proprietary development from version 6.5, announcing that it would no longer actively develop the open source project. Firebird, an open source fork of the InterBase 6 code, however, remains actively

developed. The particular choice of the servers is due to the same concurrency control mechanism, i.e. they employ Multi Version Concurrency Control to ensure snapshot isolation. In fact both products have a long history of support for multi-versioning - Borland's Interbase being one of the first systems to offer Snapshot Isolation (Thakur 1994). Although we have implemented TPC-C client for use with two commercial servers we have not focused the experimentation on these products partly because the licenses constrain the users' rights to publish performance related results. However, we have used different versions of the open-source DBMSs, both PG and FB, during the evaluation of DivRep, as well as for the selection of its components (Section 5), with the aim of gaining a thorough perception of its performance.

It should be noted that no restriction in the testbed exists regarding the choice of hardware and software. Different hardware platforms, operating systems and database servers are deployable in the testbed. The modular design of the testbed allows for the use of different client applications. Therefore, it is possible to deploy another type of benchmark, or a real-life application in order to evaluate server performances under a different operational profile. Our measurements are more detailed than those required by the standard, e.g. we record the response times of the individual SQL operations. The testbed features a set of configuration parameters that enable us to run different types of experiments. Some of the more important ones are as follows:

- *Server configuration* - We can experiment with either single servers or replicated setups. If evaluating the performance of replicated setups we can use the servers from the same vendor (non-diverse replication) or from different ones (diverse replication).
- *Number of clients (load)* – We control the concurrency degree with this parameter. Additionally, the concurrency level of the TPC-C application is configurable through *think times* parameter too (Section 2.4). Think times represent the time spent, by the operator, to read the result of the transaction at the terminal before requesting another transaction (TPC 2002a).
- *Operational profile* - We can change the probabilities for the transaction types specified by the TPC-C standard, e.g. instead of the default write-intensive profile a read-only mix can be executed.
- *Workload size* - We control the workload size by varying the number of transactions in an experiment.

4.2. Preliminary Experiments – Systematic Differences in the Performance of Diverse Servers

When DivRep is deployed, the faster responses for SQL operations might come from different replicas. The phenomenon is somewhat related to mirrored disk configurations, where one can take advantage of the random difference between the physical disks' response times to reduce the average response time on reads (Chen, Lee et al. 1994). The variability in SQL operations' durations is true for both diverse and identical server configurations. In the case when the *optimistic* regime is used, the fact can be turned into a performance improvement because the overall transaction response time might be shorter than the corresponding transaction time of the individual servers. Evidently, the observation is dependant on the type of workload (transactional mix) used and it will manifest if both servers skip (a subset of) SQL operations. Moreover the response times of DivRep deploying diverse servers might be shorter than the corresponding response times when non-diverse servers are used. This is true if *systematic* differences are observed in the response times of the diverse database servers (Gashi, Popov et al. 2004).

We conducted a preliminary empirical study to assess the performance effects of using the *pessimistic* and *optimistic* regimes of DivRep when either diverse or non-diverse servers are used and investigate differences in response time with the different setups. In this study we have used two open-source servers PostgreSQL (PG) 7.2.4 and Interbase (IB) 6.0. We used TPC-C as the client application. We used several identical machines (Intel Pentium 4 (1.5 GHz), 640MB RAMBUS RAM) with different operating systems:

- Microsoft Windows 2000 Professional for the client(s) and the IB servers.
- Linux Red Hat 6.0 for the PG servers.

The servers ran on four machines: two replicas of IB (IB1 and IB2) and two replicas of PG (PG1 and PG2). Before the measurement sessions, the databases on all four servers were populated as specified by the standard. We ran two experiments with different loads on the servers:

- *Experiment 1*: A single TPC-C client for each server.
- *Experiment 2*: 10 TPC-C clients for each server, each client using one of 10 TPC-C databases managed by the same server, so that we could measure the servers'

performance under increased load. In this way conflicts between different clients were avoided and consistency was trivially preserved.

In both experiments each client executed 10,000 transactions. Our objective in the study was not just to repeat the benchmark tests for these servers, but also to get preliminary indications about the performance of an FT-node using diverse servers, compared to one using identical servers and to a single server. Our measurements were more detailed than the ones required by the TPC-C standard. We recorded the response times for each individual transaction, for each server. We were specifically interested in comparing two architectures:

- Two *diverse servers* concurrently process the same stream of transactions.
- A reference, non-diverse architecture in which two *identical servers* concurrently process the same stream of transactions.

All four servers were run concurrently, receiving the same stream of transactions from the test harness, which produced four copies of each transaction/SQL operation. The overhead that the test harness introduces (mainly due to using multi-threading for communication with the different database servers) is the same with and without design diversity. The comparison between the two architectures is based on the *transaction response times*, neglecting all extra overheads that DivRep middleware would introduce.

We compare the performances among all four server replicas. For each pair of server replicas we calculate the minimum and maximum response time pertaining to a particular transaction by comparing the respective replica results. The former approximates the performance of the *pessimistic* regime of operation and in the same way the latter approximates the performance of the *optimistic* regime of operation, for this particular mix of transactions and setup.

We used the following measures of interest:

- *Mean transaction response times* for all five transaction types (Figure 4-2).
- *Mean response times per transaction* of each type (Figure 4-3).

Firstly we examine the results obtained with the *Experiment 1*. With *two identical database servers* (last two server pairs in Figure 4-2), the difference between the mean times is minimal, within 10%. The mean times under the *optimistic* and *pessimistic* regimes of operation remain very close (differences of <10% for IB and <15% for PG). IB is the faster server, being almost twice as fast as PG, for this set of transactions.

When we combine *two diverse database servers* we get a very different picture. Now the *optimistic* regime can deliver dramatically better performance than the faster server (IB). The mean response time is almost 3 times shorter than for IB alone (compare the first two bars for the first four pairs). When we consider the *pessimistic* regime (represented by the values of MAX(Server1, Server2)), the value of the mean response time is larger than the respective value of the slower server, PG, but the slow down is within 40% of PG's mean response time. This approximates the cost of the improved dependability assurance.

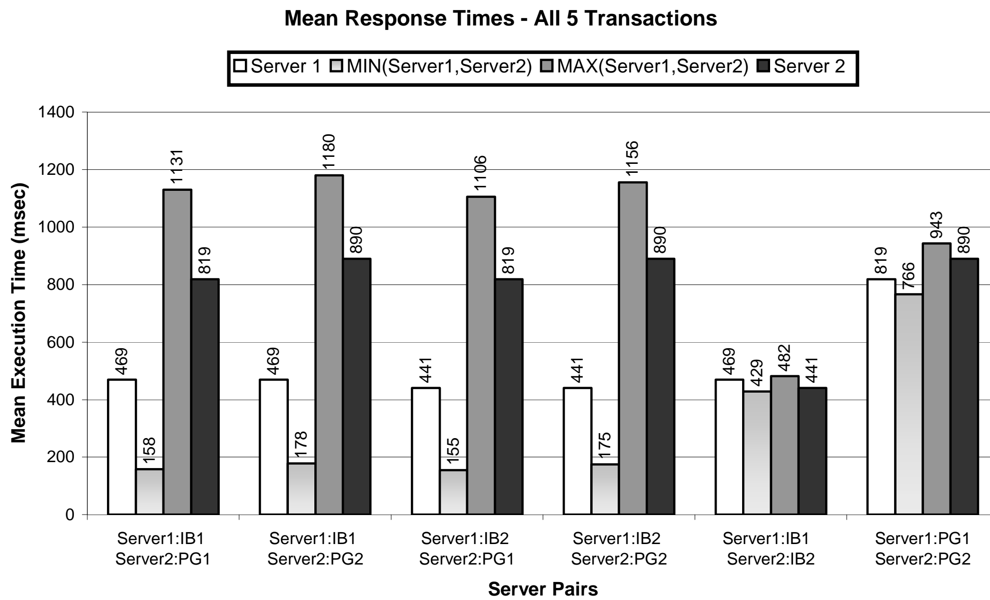

Figure 4-2 Mean response time for all five transaction types over 10,000 transactions for two replicas of Interbase 6.0 and two of PostgreSQL 7.2.4. The X-axis lists the servers grouped as pairs (Server1 and Server2). Each server may be of type Interbase (IB) or PostgreSQL (PG). For each of the 6 server pairs the vertical bars show: mean response times of the individual servers and mean response times of the approximations for the *optimistic* and the *pessimistic* regime of operation, MIN(Server1, Server2) and MAX(Server1, Server2) respectively.

In order to understand why a diverse pair is so different from a non-diverse pair we looked at the individual transaction types. The mean response times of the five transaction types individually are shown in Figure 4-3. The figure indicates that the servers "complement" each other in the sense that when IB is slow (on average) to process one type of transaction, PG is fast (*New-Order* and *Stock-Level*) and vice versa (*Payment*, *Order-Status* and *Delivery*). These *systematic* differences illustrate why a diverse pair outperforms a non-diverse one so much when the *optimistic* regime is used, and why it is worse than the slower server when the *pessimistic* regime is used (Figure 4-2).

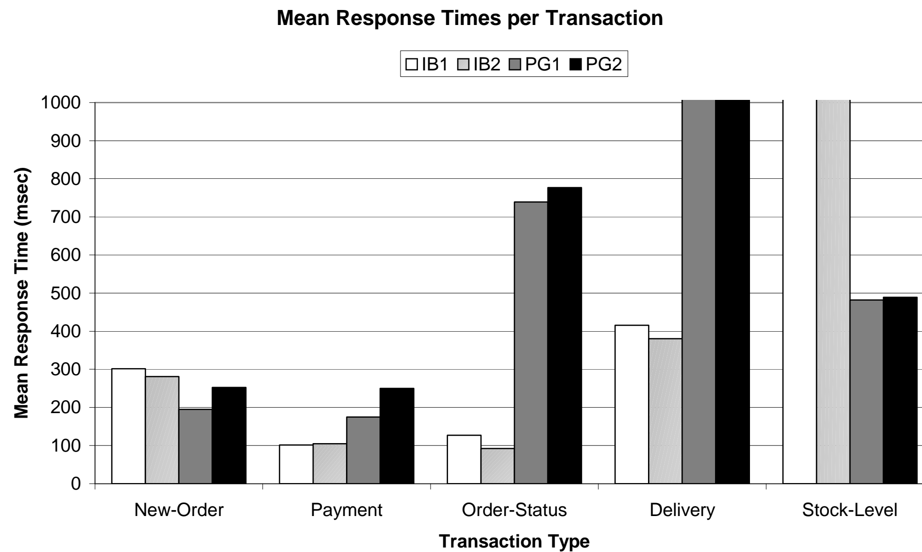

Figure 4-3 Mean response times by two replicas of Interbase 6.0 and PostgreSQL 7.2.4 for all five transaction types. The X-axis lists the transaction types (New-Order, Payment, Order-Status, Delivery and Stock-Level). The Y-axis gives the values of the mean response time in milliseconds for each of the servers (IB1, IB2, PG1 and PG2) for a particular transaction type.

In addition to the mean execution times, we have calculated the percentage of the faster responses coming from either IB or PG for each transaction. For three transaction types the situation is clear-cut. IB is always the faster server for *Order-Status* and *Delivery* transactions, while PG is always the faster server for *Stock-Level* transactions. For *New-Order* and *Payment* transactions instead, the server that is faster on average does not provide the faster response for each individual transaction. Consider the pair {IB1, PG1}. For *New-Order* transaction, PG1 is faster than IB1 on 81.2% of the transactions but slower on 15.6% (3.2% of the response times were equal). The situation is reversed for *Payment* transaction: 77.2% of the faster responses come from IB1, 15.3% from PG1. This fluctuation is further revealed in Figure 4-4. Both observations indicate that diverse servers under the *optimistic* regime would have performed better (for this transaction mix and load) than a pair of identical servers.

This pattern of the two database servers "complementing" each other was also observed in *Experiment 2*, under increased load with 10 TPC-C clients. During this experiment the servers were "stretched" so much that the virtual memories of the machines were exhausted. Similarly to the observations of *Experiment 1*, when *two identical servers* are used the difference between the mean response times is minimal, within 10%, and the difference between the mean response times of the *optimistic* and

pessimistic regime remain less than 10% for both servers. Again IB is the faster server.

The mean response times when *two diverse servers* are considered under the *optimistic* regime are around four times shorter than for IB alone. Under the *pessimistic* regime, the mean response time is of course larger than the value of the slower server (on average), PG, but the slow down is within 60% of PG's mean response time (it was 40% in *Experiment 1*, when a single client was used).

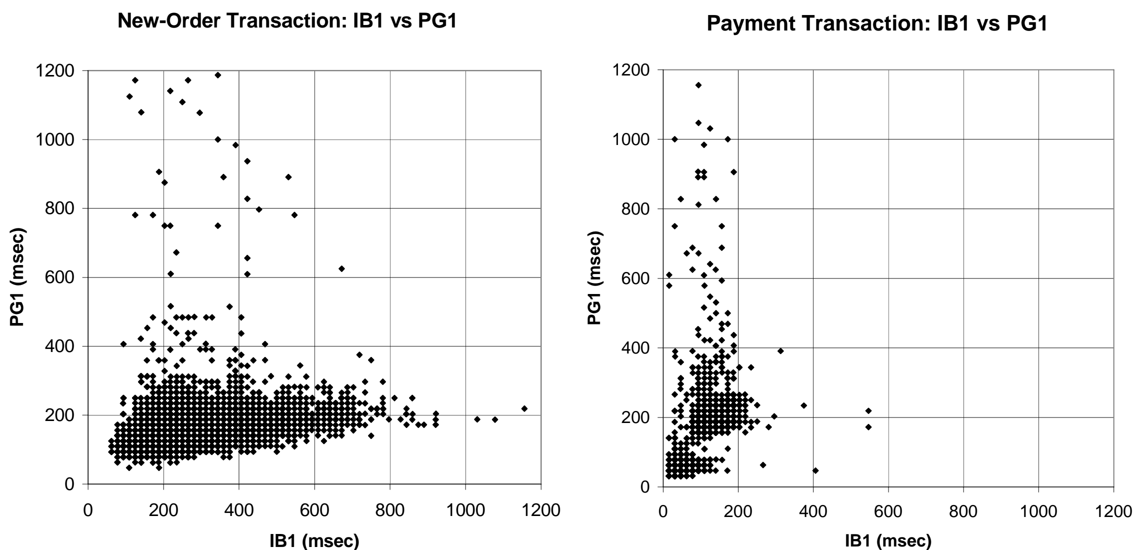

Figure 4-4 Response times for New-Order and Payment transactions. The dots represent the response times of two servers for an instance of the respective transaction type. If the times were close to each other, most of the dots would be concentrated around the unit slope (observed for the pairs of identical servers, IB1 vs. IB2 and PG1 vs. PG2). If the dots are mostly below the slope, Interbase is slower (as with New-Order). If the dots are concentrated above the unit slope – PostgreSQL is slower (as with Payment). Similar results were obtained for the other three diverse server pairs.

4.3. When Diverse Redundancy Performs Better than Non-Diverse Redundancy

In the previous section we have described a study performed to gather preliminary results on performances of diverse servers. In order to evaluate the effectiveness of diversity in a different setup we performed additional experiments. The results have been reported in (Stankovic and Popov 2006). Instead of communicating with an exclusive database, each client in the new study had communicated with the same database. In order to experiment with the database servers that exhibit comparable performance we have performed the following:

- Deployed a more recent version of the server that showed as marginally slower, PostgreSQL, in the first study. The new version used was 7.4.0 instead of 7.2.4.

- Deployed the other database server, InterBase 6.0 on the same operating system as PostgreSQL. The operating system used was Linux Red Hat 6.0 (Hedwig).

The same hardware was used as in the first study. Again, the TPC-C implementation was used as the basis for client application. We ran a set of experiments with the following server configurations:

- 1IB1PG, an FT-node with a copy of IB and PG.
- 1IB, a single replica of IB.
- 1PG, a single replica of PG.
- 2IB, an FT-node with two replicas of IB.
- 2PG, an FT-node with two replicas of PG.

Each experiment comprises the *same sequence* of 10,000 transactions and was repeated five times in order to get higher confidence in the results, as detailed below. The server machines were restarted and databases restored between the repetitions. All the measurements were associated with a single TPC-C client under different *server loads* as follows:

- No additional clients.
- 10 additional clients.
- 50 additional clients.

Whenever additional clients were deployed they executed a mix of read-only transactions (RO mix) instead of the mix of transactions recommended by the TPC-C. The RO mix consists of the two read-only transactions: *Order-Status* and *Stock-Level* of almost equal proportion. The *readers* and *writers* do not conflict in the two DBMSs, since both IB and PG feature a scheduler based on MVCC (Multi-Version Concurrency Control). Hence data consistency between the replicas is guaranteed. This was experimentally confirmed by successfully running a comparison between the databases at the end of the experiments.

The overhead that the test harness introduces (mainly due to using Java multi-threading for communication of the clients with the middleware and of the middleware with the different DBMSs) is the same irrespective whether a single or two replicas are used in the experiment. It has been measured to be negligible compared with the time taken by the respective DBMSs to process the 10,000 transactions.

4.3.1. Confidence in the Results

Each experimental setup (with a fixed configuration and load) was repeated *five times* so that we could detect significant variation between the observed results due to factors beyond our control (e.g. fragmentation of files on the servers).

Figure 4-5 shows the mean transaction times for all transactions together in a 10,000-transaction run, grouped by experiment repetitions when only a single TPC-C compliant client is deployed. There is no significant variation between the results across the repetitions. The same effect was observed for a particular transaction type too.

A similar picture, consistent across the repetitions, was established for the increased load of 10 and 50 additional clients. Figure 4-6 shows the results with 50 additional clients. The only configuration with a noticeable variation between the repetitions was 1IB. In particular, the first run is 25% faster than the remaining four in terms of the mean transaction time with all transaction types. A noticeable variation also exists between the specific transaction types, for which the percentages vary between 20% and 25%. This variation, however, does not change the ordering between the configurations.

In addition, the ordering between the configurations does not change even if we execute a different sequence of transactions. This was experimentally confirmed by using different *seed* values for the particular pseudo-random number generator to execute 10,000 transactions in different orders with either a single TPC-C compliant client or with ten additional clients.

Such consistency between the observations, in particular the fact that the ordering between the configurations remains unchanged across the repeated experiments, is the reason why in the rest of the section we compare the performances using a single run per configuration.

4.3.2. Performance Comparison of Different DBMS Configurations

To compare different DBMS configurations we used the following measures of interest:

- Mean transaction time (for all five transaction types).
- Mean transaction time for a particular type of transaction.
- Cumulative transaction time, i.e. experiment duration.

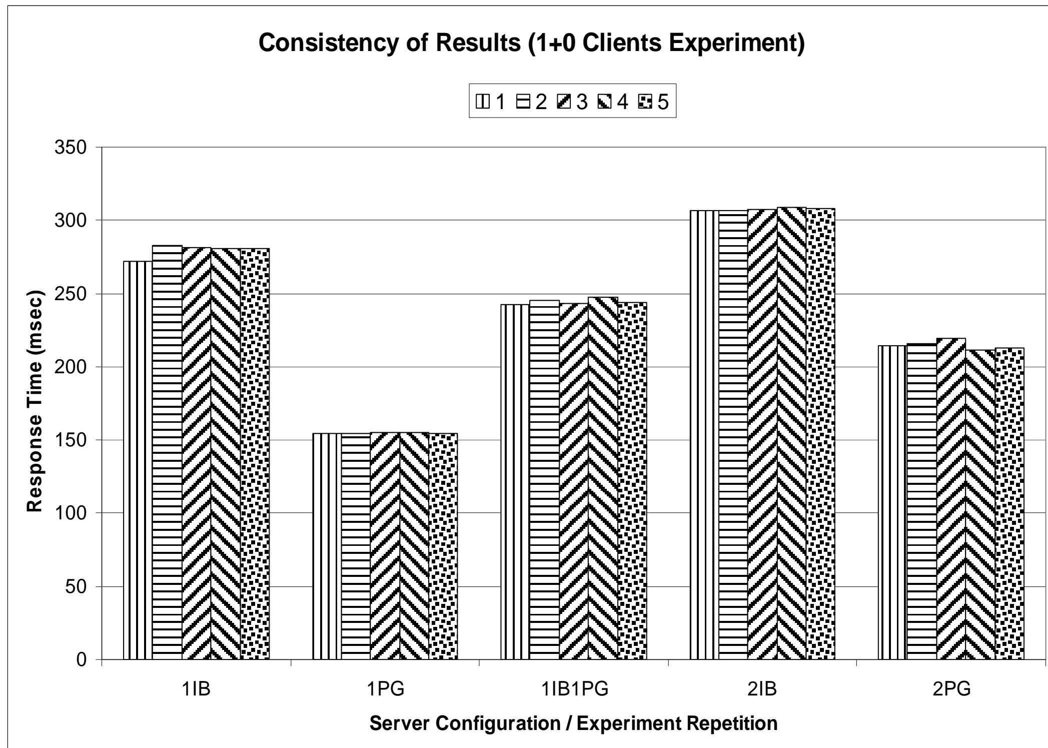

Figure 4-5 Mean transaction response times for each of the 5 repetitions and a particular server configuration (1IB, 1PG, 1IB1PG, 2IB or 2PG), when the load was generated by a single TPC-C compliant client.

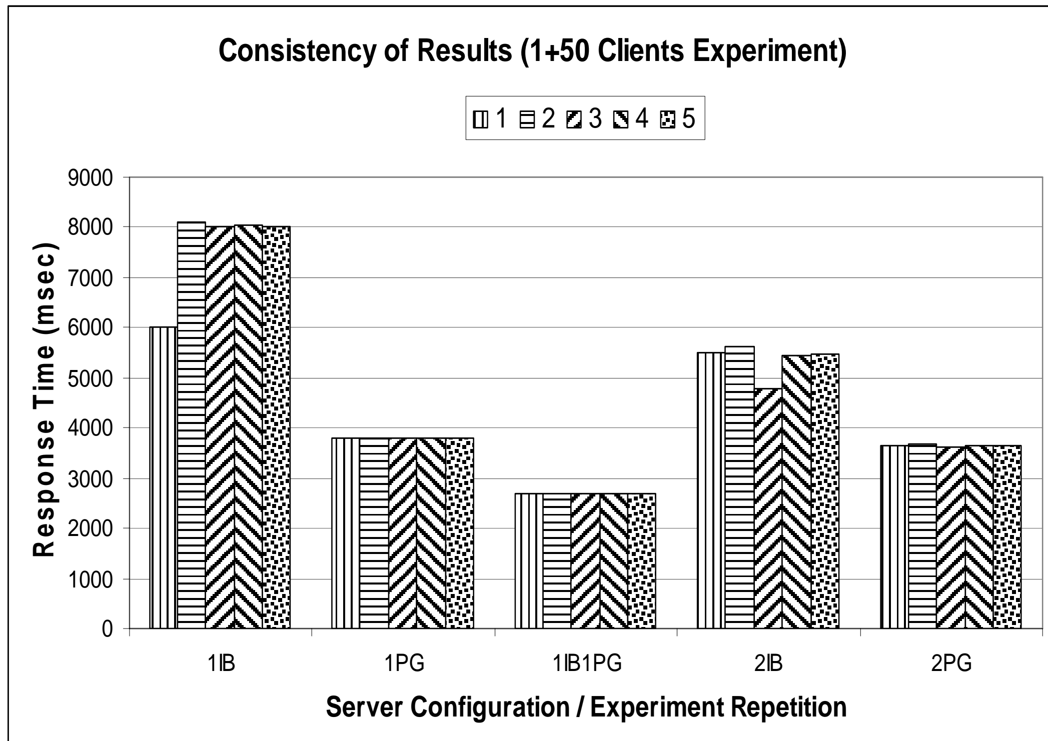

Figure 4-6 Mean transaction response times for each of the 5 repetitions and a particular server configuration (1IB, 1PG, 1IB1PG, 2IB or 2PG), under the increased load with 50 additional read-only clients.

Figure 4-7 depicts the response time when only a single TPC-C client communicates with the FT-node configurations. 1PG is on average the best configuration under this load, though transactions of type Delivery and Order-Status are faster on 1IB. The ranking changes when the load increases (Figure 4-8). Now the fastest configuration on average is the diverse pair, albeit not for all transaction types (1IB is the fastest for Order-Status and Payment, while 1PG is the fastest for Stock-Level). The figure indicates that the diverse DBMSs "complement" each other in the sense that when IB is slow to process a transaction then PG is fast (New-Order and Stock-Level) and vice versa (Payment, Order-Status and Delivery). The same observations have been recorded in the preliminary set of experiments. These systematic differences illustrate why the diverse pair, 1IB1PG, is the best configuration on average. In addition the *skip* feature enables the diverse pair to augment this advantage by omitting the read SQL operations on the slower DBMS.

Although a DBMS is fastest on average for a particular transaction type, within the transactions the fastest responses to SQL operations may come from different DBMSs. This fact is utilised by the diverse pair. Hence, it is not surprising that IB executes more SELECT operations in an experiment than PG when the two are employed as a diverse pair (IB executes 70%, while PG executes 51%). There is nothing unusual in the fact that the sum, 70% + 51%, is greater than 100%. It simply means that there are reads which are executed by both servers. If the faster server has not completed a read by the time the slower is ready to start, then both will process the particular operation. Similar results were obtained under the load with 50 additional clients.

Figure 4-9 shows how the ordering changes between the configurations as a result of a load increase. An experiment comprising 10,000 transactions under the 'lightest' load (0 additional clients) is the fastest with 1PG. Under increased load, however, the diverse pair, 1IB1PG, becomes the fastest configuration. The experiment duration with the diverse pair is shorter than with the individual DBMSs, or with either of the non-diverse (homogenous) DBMS pairs. The diverse pair is 20% faster than the second best configuration (1PG) with 10 additional clients and more than 25% faster than the second best configuration (2PG) with 50 additional clients. The benefits of the systematic difference in transaction times between the diverse DBMSs and the efficiency of the *skip* feature become more clearly pronounced when the load increases.

Figure 4-7 Mean transaction times for each transaction type and for all transactions together under a load generated by a single TPC-C compliant client. The configurations compared under this load are as follows: configurations with a single DBMS (1IB, 1PG), a configuration with a diverse pair of DBMSs (1IB1PG) and configurations with homogenous pairs of DBMSs (2IB, 2PG).

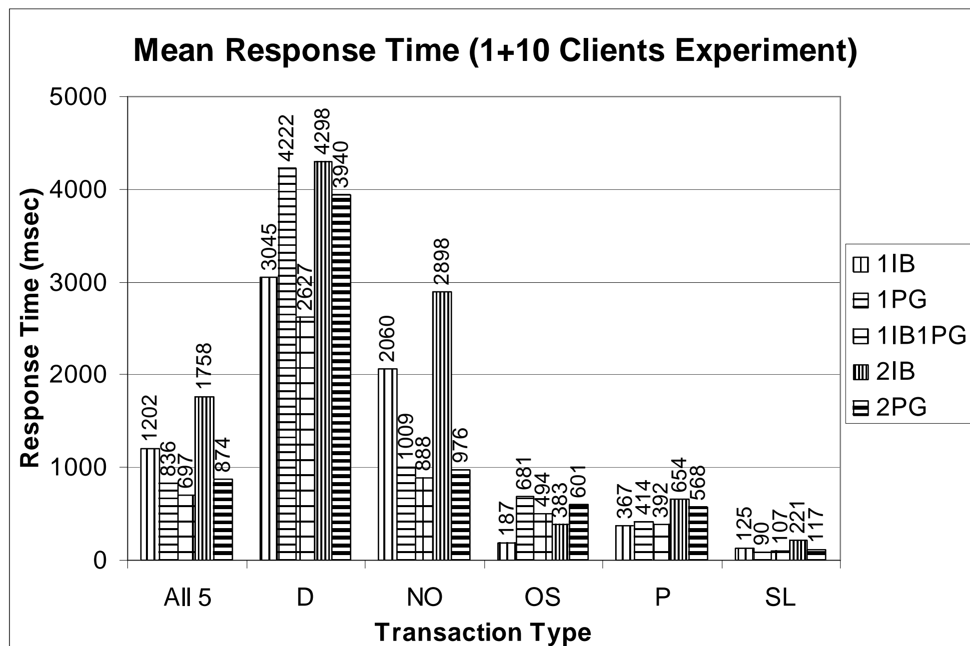

Figure 4-8 Mean transaction times for each transaction type and for all transactions together under an increased load with 10 additional read-only clients. The configurations compared under this load are as follows: configurations with a single DBMS (1IB, 1PG), a configuration with a diverse pair of DBMSs (1IB1PG) and configurations with homogenous pairs of DBMSs (2IB, 2PG).

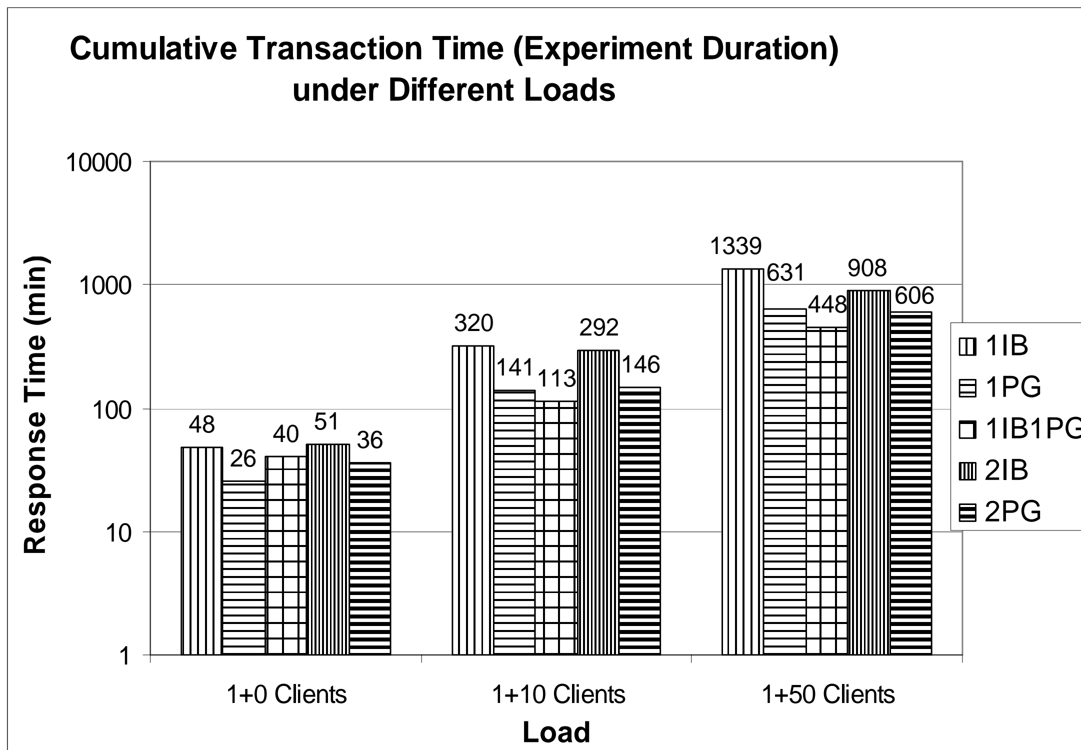

Figure 4-9 Cumulative transaction time (experiment duration) for the five DBMS configurations under different load (0, 10 and 50 additional read-only clients).

4.4. Performance Implications of Improving Dependability

DivRep middleware operating in the *pessimistic* regime improves dependability by guarding against non-crash failures in addition to tolerating crash failures. Such a degree of fault-tolerance is impossible with non-diverse solutions that deploy traditional ROWA (Bernstein, Hadzilacos et al. 1987) replication. However, the performance of the *pessimistic* regime incurs an overhead, by executing SQL operations on all replicas and then comparing the respective results; thus the performance is likely to deteriorate in comparison with ROWA. In order to evaluate the overhead, we implemented a simulation of a known middleware-based database replication solution (Lin, Kemme et al. 2005), SI-Rep, which is based on the ROWA approach. The principal goal was to use a ROWA-based replication protocol as a baseline when evaluating the performance of DivRep, and *not* directly compare the performances of the two replication schemes. The latter would be difficult to perform cleanly: the two schemes have somewhat dissimilar aims and evaluating their performances in the same framework, e.g. using the same programming language, is not easy.

4.4.1. SI-Rep Simulation

SI-Rep middleware provides replica control mechanism on top of the concurrency control offered by the underlying databases. Each of the replicas is assumed to provide snapshot isolation (Section 2.2.3) for concurrent transaction executions. SI-Rep guarantees data consistency by combining group communication primitives with the replica control system ensuring *1-copy Snapshot Isolation* (Lin, Kemme et al. 2005) in the replicated system. SI-Rep performs load balancing statically since the load is divided among all replicas. Each client submits its transactions to a *local* replica. After a transaction is executed on the local replica the *writesets* are extracted and delivered, using total order multicast, to all replicas (including the originating one). The total order multicast ensures that all conflicting writesets are validated in the same order, namely the order of writeset delivery. In the case of read-only transactions the commit follows immediately after the local execution.

Potential conflicts between the writesets, originating from concurrent transactions, are validated by the decentralized middleware. The validation proceeds in two steps, in each middleware replica. The *local* validation step, on a middleware M_x, checks for *write-write* conflicts between a local transaction, T_i, and writesets of concurrent remote transactions (the check against possible conflicts with local concurrent transactions is performed by the database replica). If the local validation succeeds, T_i's writeset is multicast, otherwise T_i is aborted. However, the validation does not stop here. The reason is that the middleware replicas might send the writesets concurrently and a number of writesets, received between sending and receiving T_i's writeset, would not be validated by M_x. These writesets are validated as part of the *global* validation step using the writeset delivery order imposed by total order multicast.

While DivRep imposes equivalent order of begins and commits on the replicas, effectively guaranteeing that a client reads the same snapshot of data from any replica, SI-Rep allows for a certain degree of concurrency of transaction boundaries. Transaction begins run concurrently on different replicas in SI-Rep since reads are not performed on all replicas. Their order relative to commit operations is potentially different at different replicas, e.g. two concurrent transactions T_1 and T_2, executing on two replicas R_x and R_y, are allowed to produce the following schedules of transaction boundaries: b_1, b_2, c_1, c_2 on R_x and b_1, c_1, b_2, c_2 on R_y.

In our simulation of SI-Rep, referred to as SimSI-Rep in the rest of the document, we have ignored the group communication system and the overhead it might incur. As in DivRep, the total order of transaction executions was established using *tb_mutex* (Section 3.1.1). The same Two-Phase Commit protocol, 2PC-DR (Section 3.1.1), which we used in DRA, is used in the simulation too - it guaranteed the transaction termination on the replicas. We have simulated the static load balancing mechanism by dividing the load *equally* between deployed replicas, i.e. a DivRep experiment with 2 replicas interacting with 50 clients was compared to a SimSI-Rep experiment, in which each replica was assigned 25 clients. Although a replica executes the read-only load of only half of the clients, in a SimSI-Rep experiment, it executes the write operations generated by all the clients. However, instead of propagating the writesets in the end of transaction (*deferred writes*) to the remote replicas we have executed the *updates in parallel (immediate writes)* on both replicas. This is a deviation from the original implementation of SI-Rep and it has implications on the comparison between two schemes. On the one hand, the execution of the SQL operations might be more expensive than the application of writesets. On the other hand, the retrieval and propagation of the writesets, the essential components of SI-Rep, are eliminated. As a result, and in addition to the lack of group communication, the simulation of SI-Rep might be favoured over DivRep. When executing read-only transactions with SimSI-Rep we have ignored the use of *tb_mutex* for the serialisation of the commits. This is because the position of the commit operations of the read-only transactions is irrelevant since no writeset is associated with them. Similarly, since readsets of the remote transactions are empty in SI-Rep, we discarded the synchronisation of their begin operations.

4.4.2. DivRep vs. a ROWA-Based Replication (SimSI-Rep)

Experimental Setup and Results

To experimentally compare the performance of DivRep against the simulation of SI-Rep scheme we again used the TPC-C implementation (Section 4.1). In the study we used newer versions of both database servers: Firebird 2.0.1 and PostgreSQL 8.1.5, referred to as FB and PG, respectively. The client and the logging machines ran Windows 2000 Professional sp4 operating system, as in the previous experiments,

while the two database servers ran atop Fedora Core 6. The hardware specifications were as follows:

- *Client machine*: 1.5 GHz CPU, 1GB RAMBUS RAM and 20GB 5400 rpms IDE disk.
- *Logging machine*: 1.5 GHz CPU, 512MB RAMBUS RAM and 40GB 5400 rpms IDE disk.
- *Server machines*: 1.5 GHz CPU, 640MB RAMBUS RAM and 80GB 7200 rpms IDE disk. The database servers were deployed using faster, more recent HDDs than in the previous experiments, i.e. we used Seagate Barracuda IDE HDDs (ST-380011A) instead of Maxtor DiamondM IDE HDD with 40GB and 5400 rpms.

Initially we have performed various experiments with the original TPC-C profile and database load. For example, we have deployed a TPC-C database with 20 warehouses in order to run experiments with varying number of clients. Due to the limited capabilities of our data storage, we observed an I/O bottleneck. Therefore, the CPU was underutilised and the throughput dropped to only several transactions per minute. We had decreased the *think times* to increase the concurrency, but the change resulted in a high conflict rate. In order to increase throughput and minimise the abort rate we changed the transactional profile to a *read-oriented* one. The read-only transactions, OS and SL, were made the most frequently executed ones; amounting to approximately 86% of executions, while each of the modifying transaction types (D, NO and P) amounted to at least 4% of all executions. Clearly, executing the read-oriented mix reduces the likelihood of conflicts. Although the *optimistic* response regime might consequently exhibit improved performance, the workload is advantageous for SI-Rep – the ROWA-based replication will improve the performance by distributing the reads among the deployed replicas. Despite the frequencies of transaction types being altered, the representativeness of the modified workload still holds, i.e. activities of an order-entry system are modelled. In order to measure the impact of high concurrency, we have changed the values of the *think times* – the mean values of think times' distribution were decreased by an order of magnitude compared with the values specified in the standard.

We have performed several types of experiments to test the performance of DivRep. The following replication configurations were used in the experiments:

- DivRep middleware running in the *pessimistic* regime (*1FB1PG-Pess.*).
- DivRep middleware running in the *optimistic* regime (*1FB1PG-Opt.*).

- DivRep middleware running in the *optimistic* regime and using a pair of non-diverse servers (*2PG-Opt*).
- Simulation of SI-Rep middleware using a pair of PG servers (*2PG - SimSI-Rep*).
- Simulation of SI-Rep middleware using a pair of FB servers (*2FB - SimSI-Rep*).

The load on the servers varied by changing the number of clients: 25, 50, 100 or 200 clients were deployed. The different loads have been used in conjunction with the read-intensive profile. Each test consisted of 50,000 transactions. Moreover, experiments with 20 clients executing the original, write-intensive workload specified by the TPC-C were performed (this is the only load under TPC-C profile for which we have not observed I/O bottleneck and the conflict rate was small). The mean of the think times distribution was decreased an order of magnitude as in the experiments with read-oriented profile. In the tests with TPC-C compliant profile 150,000 transactions were executed in each experiment. Despite taking detailed experimental logs (e.g. we recorded response times of individual SQL operations) we have chosen experiment duration, a throughput statistic (it is directly proportional to *tpmC,* a metric proposed by TPC-C standard), and average transaction response time to be the measures of interest. We ran the same set of experiments with SimSI-Rep as with DivRep. Each type of experiment was repeated at least five times to get higher confidence in the results. In the rest of the chapter it is assumed that one FB and one PG replica are deployed in DivRep middleware if not stated otherwise.

Figure 4-10 contrasts the experiment duration for DivRep middleware, running in either the *pessimistic* regime (*1FB1PG-Pess.*) or the *optimistic* regime (*1FB1PG-Opt.*), and SimSI-Rep middleware, when either Firebird replicas (*2FB - Sim-SIRep*) or PostgreSQL replicas (*2PG - Sim-SIRep*) are used. In order to compare the two replication schemes more fairly we deployed the same, non-diverse, servers in both. We compared the performances of DivRep with two PGs (*2PG – Opt.*) against SimSI-Rep with two PGs (Figure 4-11). Details about the response times of different replication schemes are given in Table 4-1.

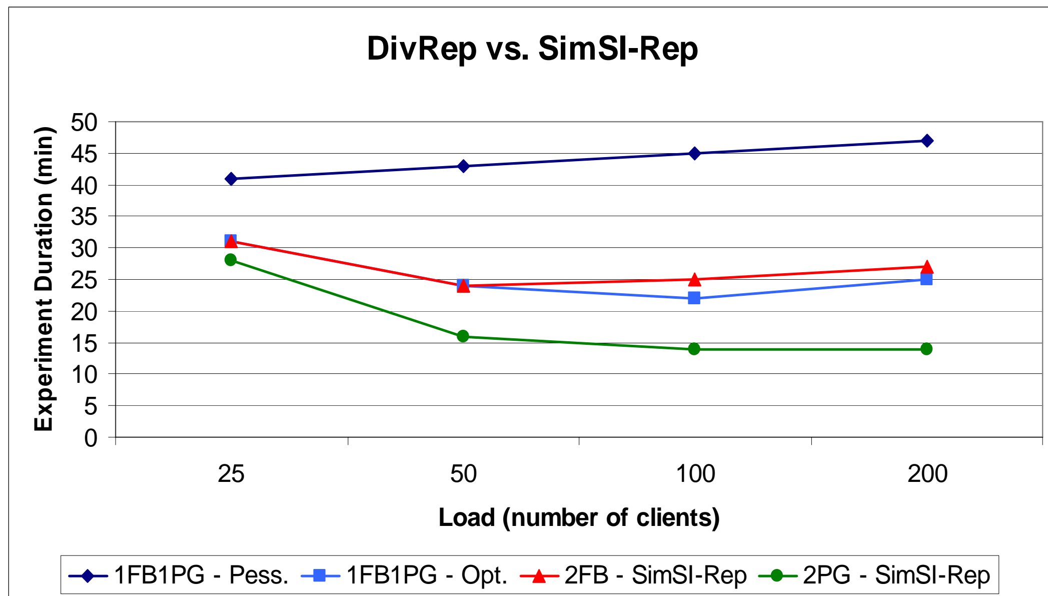

Figure 4-10 Experiment duration of DivRep running in either the *pessimistic* regime (1FB1PG - Pess.) or the *optimistic* regime (1FB1PG - Opt.) and the simulation of SI-Rep with FB (2FB - SimSI-Rep) and PG (2PG – SimSI-Rep) under different loads.

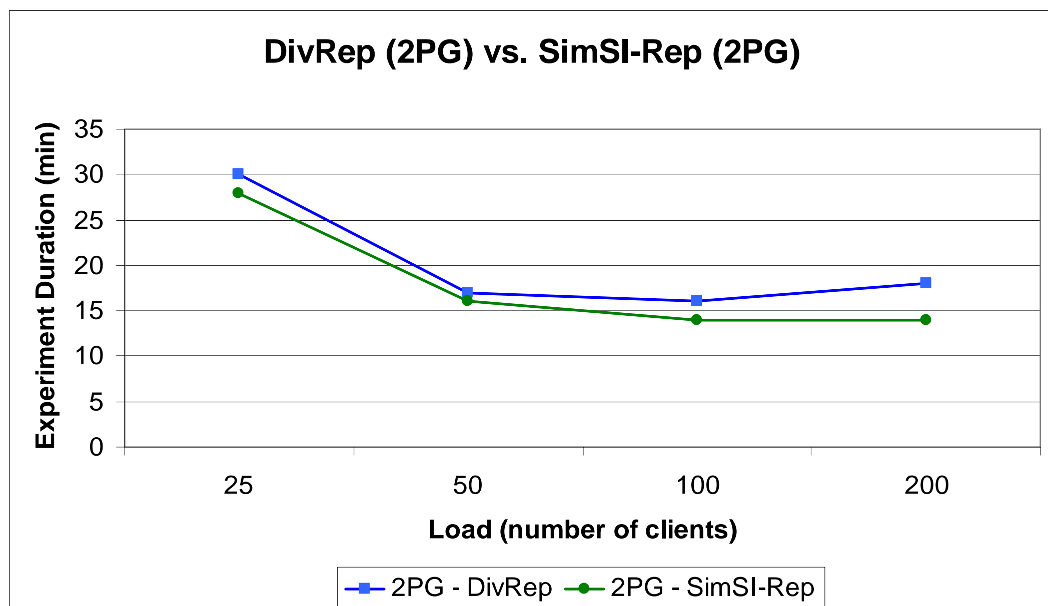

Figure 4-11 Experiment duration of DivRep with two PG servers running in the *optimistic* regime (2PG – Opt.) and the simulation of the SI-Rep with two PG servers (2PG - SimSI-Rep) under different loads.

Table 4-1 The average transaction response times for different replication schemes.

Load	Scheme Type	Response Time (ms)
25	1FB1PG-Pess.	466
	1FB1PG-Opt.	161
	2FB - SimSI-Rep	168
	2PG - SimSI-Rep	65
50	1FB1PG-Pess.	1812
	1FB1PG-Opt.	813
	2FB - SimSI-Rep	958
	2PG - SimSI-Rep	227
100	1FB1PG-Pess.	4048
	1FB1PG-Opt.	2027
	2FB - SimSI-Rep	2303
	2PG - SimSI-Rep	925
200	1FB1PG-Pess.	8921
	1FB1PG-Opt.	4251
	2FB - SimSI-Rep	4319
	2PG - SimSI-Rep	2172

Discussion of the Results

When we look at Figure 4-10 we can observe that the performance of DivRep in the *pessimistic* regime is worse than the performance of SimSI-Rep with either of the servers. Also, SimSI-Rep with PGs is superior to the performance of its FB counterpart. When the load is small (cf. 25 Clients in Figure 4-10) SimSI-Rep deploying two FB replicas is 25% faster than DivRep and SimSI-Rep with PGs is more than 30% faster. We observed that the CPU utilisation on the machine running FB server was moderate under this load while PG demonstrated very low CPU activity. As the load increases, the performance of *1FB1PG-Pess.* degrades, while SimSI-Rep exhibits a faster result (experiment duration decreases) by sharing the load among replicas and utilising the spare CPU capacity. Hence the difference between *1FB1PG-Pess.* and SimSI-Rep increases: the result of the 2PG SimSI-Rep experiment is more than 70% faster than DivRep in the *pessimistic* regime under the highest load (200 clients). It is worth noting that the performance of SimSI-Rep with FB deteriorates under the highest load. The CPU becomes a bottleneck and consequently the experiment duration is longer. This can be partly explained by different architectures of the two servers: PG uses shared memory for multiple server backends, i.e. processes, while FB (and in particular the architectural model we used in the experiments, i.e. *Classic server*) allocates fixed amount of memory to each connection for individual use.

The results in Figure 4-10 show how the two schemes compare under the *read-intensive* profile. A similar ratio between DivRep and SimSI-Rep, as in the experiments under the read-oriented profile with 25 clients, was observed with the experiments when the proper TPC-C profile and the load of 20 clients was used: the SimSI-Rep deploying two FB replicas was again 25% faster than *1FB1PG-Pess.* and the SimSI-Rep with PGs was around 35% faster.

The difference between DivRep in *pessimistic* regime and SimSI-Rep configurations is significant but it can be reduced if DivRep is deployed in the *optimistic* regime of operation (see the blue line with the square marker in Figure 4-10). We can see that SimSI-Rep with two PG servers is still the best configuration, although the difference is less pronounced. The 2PG SimSI-Rep is superior under the highest load (cf. 200 Clients), when it is approximately 45% faster than *1FB1PG-Opt*. On the other hand the SimSI-Rep with FB is never better than the *1FB1PG-Opt*. Under the highest load, both DivRep and SimSI-Rep with FB exhibit poor performance because the CPU load on the FB replicas becomes the bottleneck.

As expected, the reason for the improved performance of DivRep in the *optimistic* regime is a form of *dynamic* load balancing achieved using the *skip* feature. In contrast to the *pessimistic* regime the reads are not necessarily executed on every replica. Moreover, the faster responses might originate from any of the replicas. This is intensified with the use of diverse servers where systematic difference in duration of SQL operations might be observed. The experimental logs confirmed this premise: in the *1FB1PG-Opt.* experiment under the load of 200 clients, approximately 58% of SELECTs were skipped, i.e. they were executed only on a single replica ((100 – 80124/187882)%).

Table 4-2 Efficiency of *skip* feature.

Count of SELECTs on PG	165861
Count of SELECTs on FB	102145
Count of SELECTs on both	80124
Total count of SELECTs	187882

On the other hand, when the *pessimistic* regime is used, the systematic differences have a negative impact on the performance. Both servers execute all operations and both produce slower responses and, thus, the performance of DivRep is likely to be

worse than the performance of the individually slower server (FB). One might want to know if the better performance of the *optimistic* regime is caused, at least partially, by the lack of error detection offered through the Comparator function. To evaluate this assumption we have performed experiments in which the *pessimistic* regime of operation was altered so that no comparison of the results was performed, though both diverse replicas executed all operations. These experiments showed no significant difference from the experiments with the fully featured *pessimistic* regime in place, i.e. when the Comparator function was used the experiment durations, for the different loads, ranged from 40 to 45 minutes. We acknowledge the possibility that the result is application specific, i.e. if instead of the TPC-C client, an OLTP workload, an OLAP (On-line Analytical Processing) application was used for experimentation, e.g. TPC-H benchmark (TPC 2007), where complex SELECT operations return possibly large result sets, the error detection mechanism could have more of an impact. However, comparing the hash values of the result sets (Section 3.1.1), instead of their exhaustive value-by-value comparison, would alleviate the drawback.

The difference between DivRep and SimSI-Rep is very significant if we look at the average transaction response times for each scheme under the read-oriented profile (see Table 4-1). The average response time of SimSI-Rep with two replicas of PG is less than 15% of the average response time observed with DivRep running in the *pessimistic* regime under the load of 25 and 50 clients. The difference, however, decreases with the load increase. SimSI-Rep with two PGs is faster than DivRep deployed in the *optimistic* regime, too, although the difference is less pronounced. The superiority of SimSI-Rep is expected since, as pointed out above, an extreme form of optimisation is applied to SI-Rep at the expense of limited dependability assurance. However it is also due to the differences in performance of the two diverse servers deployed in DivRep. This is confirmed with the comparison between SimSI-Rep with two FBs and DivRep in the *optimistic* regime – the latter is always faster.

With the above evaluation we compare the performance of different replication schemes. However the comparison is somewhat blurred by the significant differences in performance of the individual servers. Certainly, the difference between DivRep and SimSI-Rep would not be the same if we observed different performances of the individual servers. That is why we have performed the experiments with DivRep deploying a pair of the marginally faster server (PG) and compared it with the results of the SimSI-Rep scheme using the same server (Figure 4-11). In this way we have

eliminated the different servers as a source of variable performance. We can observe that the performances of the two replication schemes differ only marginally. SimSI-Rep is at most 20% faster than DivRep. This difference is observed under the highest load with 200 clients. The reason for such a discrepancy is that the load balancing is more effective with SI-Rep and as a result extra CPU cycles can be spent on parallelizing clients' requests. The dynamic load balancing with *2PG – Opt.* is ineffective because the non-diverse replicas *do not exhibit systematic differences*, the *skip* feature is not utilised very often and most read operations are executed by both replicas.

To further scrutinise the considerable differences in individual server performances, and evaluate the overhead of the synchronisation introduced by the replication middleware, we performed a baseline comparison between DivRep and non-replicated server configurations. One of the goals of the experimentation was to check if performance penalty exists (and if it does what is its magnitude) when DivRep is used. To obtain the results for a non-replicated solution, with either one FB server or one PG server, we have excluded the replication protocol (no acquisition of the global mutex (*tb_mutex)* or 2PC-DR protocol was performed), and instead we let the particular DBMS impose the order of transactions. We aborted and repeated the transactions for which the DBMS had reported concurrency conflict exceptions. The distinction between *optimistic* and *pessimistic* regime of operation is irrelevant with non-replicated solutions and, thus, only one experiment type was performed. We used the read-oriented profile under the load of 100 clients. The experiment durations, for the two non-replicated configurations, are as follows:

- 1FB experiment: 45 minutes.
- 1PG experiment: 12 minutes.

Clearly, the difference between the performances of the individual servers is considerable. The results show that the slowness of FB determines the performance of DivRep – the duration of the 1FB experiment is the same as the one of DivRep employing the diverse pair and executing in the *pessimistic* regime. The extra dependability assurance achieved by results comparison in DivRep comes at no cost since the performance of 1FB is poor. The overhead of the replication middleware is insignificant.

To get more accurate result about the replication overhead we compare the performance of the faster server, PG, against DivRep solution deploying the non-

diverse pair, 2PG, executing in the *optimistic* regime (thus, no slowness of one of the servers in DivRep blurs the evaluation of the replication overhead). The difference is more pronounced now – the overhead due to the serialisation of DivRep is 1/3 (12 min. for 1PG experiment vs. 16 min. for 2PG DivRep experiment (Figure 4-11)). The reasons that justify the choice of the *optimistic* response regime in comparison with the non-replicated solution are as follows:

- The performance of the 2PG DivRep in the *optimistic* regime is similar to the performance of the non-diverse server configuration in the *pessimistic* regime – the performance boost due to *skip* feature is ineffective when a non-diverse pair is deployed.
- The difference that remains is the results' comparison. However this feature is confounding in the evaluation of the performance overhead introduced by DivRep – it is not a necessary element of the replication protocol.

The difference in performance of the replicated and non-replicated server configuration can be partly attributed to the specifics of the workload – the difference would have been less apparent if the transaction durations were longer relative to the DivRep replication overhead. Had we executed a workload characterised with longer running transactions, in which a transaction lasts significantly longer than the corresponding processing needed for the proposed replica control in DivRep, the contention for the global mutex (*tb_mutex*) would have been less frequent and the abovementioned overhead of 1/3 could have been smaller.

4.4.3. Discussion of DivRep vs. SimSI-Rep Comparison

Although the empirical evidence presented in the previous subsection shows that dependability assurance via design diversity might be expensive we would like to point to a couple of important aspects.

Firstly, though performance penalty may appear excessive there are ways of reducing it by deploying *optimistic* regime of operation of the middleware. In contrast with other optimised schemes (e.g. the known ROWA replication) such an approach is not merely sacrificing the dependability assurance, but can be *confidence* based, as explained in Section 3.3. Executing the same SQL operation, with different parameter values, *sufficiently* many times in the *pessimistic* regime and observing no disagreement between replicas would gradually build the confidence that the subsequent instances of the operation can be executed in the *optimistic* regime of

operation.

Secondly, we concluded that the significant performance deterioration recorded in the study can be attributed to the significant difference in the individual performances of the servers with the particular application profiles. In all reported cases with the two servers the chosen version of FB (2.0.1) turned out to be significantly slower in comparison with the chosen version of PG (8.1.5). The replication middleware itself (DivRep) is hardly an issue as it is only marginally slower in comparison with SI-Rep when used with two replicas of the faster server (PG) (Figure 4-11). It seems that the comparison DivRep vs. SI-Rep, when both replication schemes use two replicas of PG, favours the latter. This is because when two identical replicas are used with DivRep there are *no systematic differences* in the speed of processing the read operations by the deployed replicas. As a result, under DivRep each of the replicas will process a significantly larger proportion of *all read operations* generated by *all connections* than only half of the reads (all reads generated by half of the connections) it would under SI-Rep. This last observation just reiterates how important the individual performance of the diverse servers is - the results clearly indicate an important aspect for 'optimal selection' for diversity (we provide a formal approach for the selection in Section 5). Minimising performance cost is a factor, which may significantly affect the selection. We back up the assertion with the following result. We repeated an experiment with DivRep in the *optimistic* regime using a commercial server, instead of Firebird, in the diverse pair. We had only a trial version of the commercial server on our disposal and, thus, only 50-client experiment was performed. The experiment duration of DivRep in the *optimistic* regime was 18 min. – this is a significant improvement compared to 24 min. it took *1FB1PG-Opt*. (Figure 4-10) to complete the same type of experiment.

The particular ratio between the performance of DivRep with an FB and a PG and SimSI-Rep with two PGs was recorded under the specific experiment parameters (TPC-C profile and the read-oriented profile derived from it by altering the frequencies of the transaction types). One can envisage a range of alternative profiles (applications, configurations, etc.), on which the servers may behave differently, e.g. by becoming individually close in terms of performance (score individually closely on the chosen profile). For example, TPC-W (TPC 2002b) workload could have been used in the experimentation. Under such regimes the difference between the replication schemes compared here (DivRep vs. SI-Rep) may change: they may get

closer or even the ordering may change, e.g. DivRep may outperform SI-Rep. The rational for such an expectation is as follows. Suppose we have a server X and a server Y, which for the chosen profile are comparable; but such that X is much faster than Y on say 50% of the read operations, while Y is much faster than X on the other 50% of the read operations. Further assume that the reads are arranged in batches which are read faster by the same server ('X batch' refers to reads on which X is much faster than Y, while 'Y-batch' is a batch on which Y is much faster). Under this new assumption, in extreme cases, Y will do only the first read of the 'X-batch' and skip the rest of the batch, while X will do only the first read of the 'Y-batch' and skip the rest of it. Clearly, under such a hypothetical arrangement not only will the fastest response on the read operations be received faster than the response from SI-Rep (no matter whether with X or Y servers), but also the transactions by the diverse pair in DivRep will take shorter time to complete than the transaction by SI-Rep. Indeed, the servers will almost only read their own batches in full and skip the batches of the other server. The load generated for the servers by the reads under DivRep will then be almost 50%, e.g. identical to the load under SI-Rep. In both replication schemes, the servers execute all writes. Thus, it is plausible to expect that if the assumptions are satisfied DivRep will be faster than SI-Rep. Whether such a hypothetical scenario will ever materialise is (so far) unresolved; we failed to find a profile, derived from TPC-C, which would demonstrate this possibility.

4.4.4. User-Centric Analysis

In order to further scrutinise the performance implications of DivRep and extend the comparison with SI-Rep scheme we have performed an additional, *user-centric*, analysis. The analysis centres upon the user's perspective of transaction response times. In DivRep scheme, the replication is performed on the level of SQL operations. Recall from Section 3 the interaction between clients, middleware and replicas. A client, i.e. an emulated user, sends an SQL operation to the middleware and the middleware forwards it to the replicas for execution. When the faster result is received the middleware relays it back to the client. Once the client has received results for all SQL operations of a transaction, it sends the *commit* operation to the middleware. After the replicas have performed the commits and the middleware reported the outcome to the client, a new transaction can start. This is an inherent characteristic of eager database replication and an atomic commitment protocol, of which 2PC (Gray

1978) is the simplest and best-known example. In DivRep, due to the different pace of processing of SQL operations by the diverse replicas, the client perceives extended commit time because, apart from the genuine durations of the commit executions on the replicas, they include the time needed for the slower server to "catch up", i.e. execute the remaining SQL operations. Performance could be improved if the client regarded an alternative event as the end of the transaction – shorter response times would be observed if the delivery of the result of the last DML operation was considered as the transaction end. The transaction response time calculated in this way is the focal point of the user-centric analysis. Although the response time observed by the client might be improved in this case, the throughput would remain the same because, if the consistency is to be preserved, the client needs to wait for the slower server to execute all SQLs in a transaction before it starts the consecutive one.

In TPC-C benchmark, like in many real-world applications, the execution of successive transactions by the same user is separated by a time delay. These delays, so called *wait times* to use TPC-C terminology, represent the time needed for the user to read results of the last transaction (*think times*), select the type of the following transaction and enter the required parameters for its execution (*keying times*). The characteristic of OLTP workloads that each client waits in between the execution of consecutive transactions might be beneficial for performance of DivRep. The middleware could notify the client that the transaction has been completed as soon as the faster of the replicas produces the result of the last DML operation in a transaction. The client would start *waiting* immediately following the notification, instead of postponing it until both servers have completed. However the commit would not happen until both replicas are ready to do so, i.e. both have executed the operations. The *early waiting* would compensate for the tardiness of the slower server and the throughput of DivRep might increase. Nonetheless two possibilities require further scrutiny.

The first one regards the ratio between the wait time and the time needed for the slower server to catch-up. A performance improvement would be observed only if the wait time is longer than the time needed for the slower server to finish the transaction. Otherwise a performance penalty would be incurred since the "catching-up" time would delay the start of the subsequent transaction (Figure 4-12).

Secondly, is it possible that the slower server detects a concurrency conflict during the catching up phase, and how should DivRep deal with it? If such an event happens the

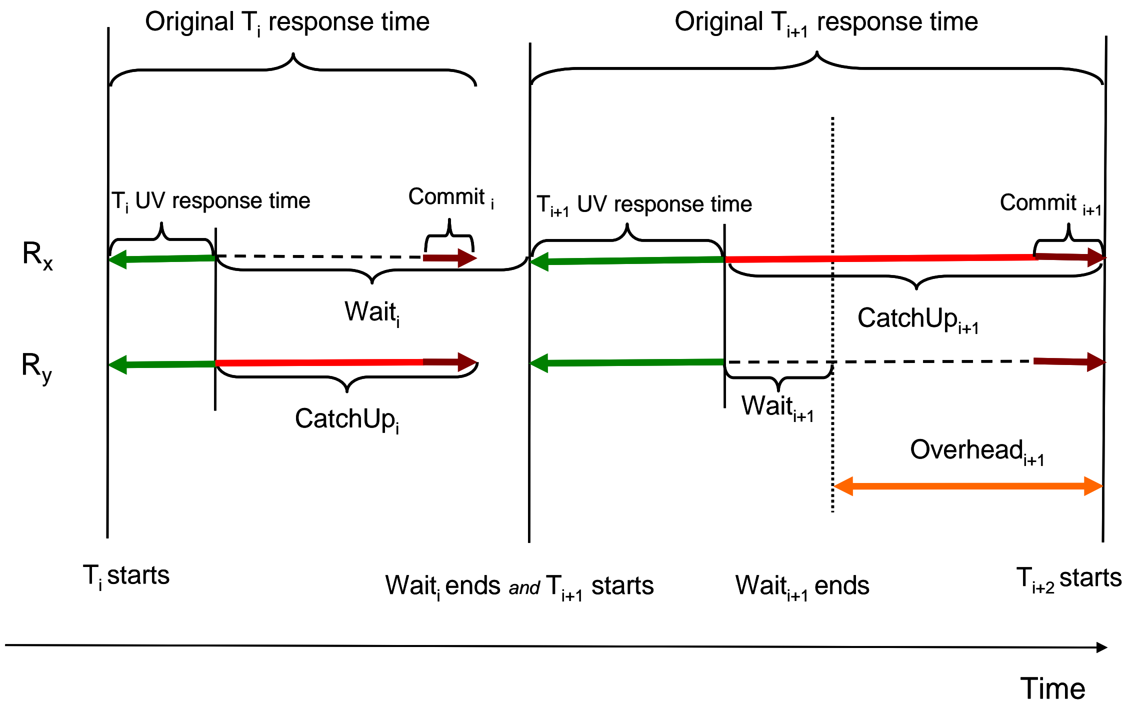

Figure 4-12 The execution of two transactions, T_i and T_{i+1}, on two replicas R_x and R_y. The following measures are depicted: the *original* transaction response times, respective wait times ($Wait_i$ and $Wait_{i+1}$), CatchUp times ($CatchUp_i$ and $CatchUp_{i+1}$), commit times ($Commit_i$ and $Commit_{i+1}$), *User-view* transaction response times (T_i UV response time and T_{i+1} UV response time) and (possible) overhead ($Overhead_{i+1}$). T_{i+1} starts without delay once $Wait_i$ has completed, while T_{i+2} has to wait until $CatchUp_{i+1}$ has finished and thus an overhead, $Overhead_{i+1}$, is incurred. *Waiting* ($Wait_i$ and $Wait_{i+1}$) commences immediately after the respective UV transaction times (T_i UV response time and T_{i+1} UV response time) end. Note that the commit of a transaction is depicted as the sum of the maximum of commit durations of the two replicas, e.g. MAX($Commit_i(R_x)$, $Commit_i(R_y)$), and the time spent on acquisition of *tb_mutex* (Section 3.1.1). The interaction between the replicas and the middleware and, similarly, between the middleware and the client, is omitted in the figure for clarity.

user would have to be notified that the middleware had falsely reported successful transaction end. Although the client logic might have to be changed accordingly, no reconciliation techniques are necessary on the replicas, because DivRep uses eager replication and a variant of 2-Phase Commit, i.e. no replica would unilaterally abort or commit a transaction. Most importantly, it is *impossible* that a concurrency conflict is detected *only* when the slower server executes in the catching up phase. The conflict must have been already reported by the faster server. This is true because replica determinism is guaranteed by DivRep - transaction boundary operations are serialised and *2-Phase locking* mechanism, and in particular *version-creation-time conflict check* (Section 2.2.3), of the replicas is employed for the identification of *write-write* conflicts. As a result, all conflicting operations are detected on both replicas. This behaviour is ensured by both *pessimistic* and *optimistic* regime of DivRep. Even if replicas omit some read operations in the *optimistic* regime, it is ensured that a concurrency conflict would be detected before the catching up phase, given that an

operation has been executed on at least one replica and only *write-write* conflicts are possible.

Since the client is notified of a successful transaction end once the faster server finishes the last DML operation, one might think that, when placed in the user-centric perspective, DivRep resembles lazy replication solutions. This is, however, not true because the execution of updates on both replicas is performed as a part of a single transaction using 2PC-DR protocol (Section 3.1.1) i.e. no separate transactions are started for propagation of updates to the remote replica and database states on the replicas never diverge.

We have used the set of experiments already described in Section 4.4.2 to evaluate the performances of different replication schemes in regard to the user–centric approach. In this section we are focusing on the comparison between three types of replication:

- DivRep scheme operating in the *optimistic* regime and employing a diverse pair of servers, DivRep 1FB1PG.
- DivRep scheme using the marginally faster server, DivRep 2PG, while operating in the *optimistic* regime.
- SI-Rep scheme using the marginally faster server, SimSI-Rep 2PG.

Evidently, user centric analysis has little meaning for centralised database systems and thus we focus on replicated solutions. Each of the replication schemes are scrutinised using three different load/workload combinations:

- *Experiment 1*: 20 clients executing the write-intensive profile specified by TPC-C.
- *Experiment 2*: 100 clients executing the read-oriented profile (Section 4.4.2) consisting of TPC-C transactions with modified frequencies.
- *Experiment 3*: 200 clients executing the read-oriented profile consisting of TPC-C transactions with modified frequencies.

Two measures are of main interest to us:

- *User-view (UV)* transaction response time (Figure 4-12), of which end timestamp is the point in time when the client received the result of the last DML operation. Clearly, the result is sent by the faster replica. The begin timestamp remains the same as the one recorded for the *original* transaction response times when user-centric analysis is not conducted. It is the timestamp taken before sending the *begin* operation to the replicas. In the rest of this section both *UV transaction response time* and, just, transaction response time, are terms used to denote the measure.

- *Overhead*, which represents the delay incurred in the event of catching-up time being longer than the corresponding wait time:

$$\begin{aligned}&\mathit{if}\ (CatchUp_i > Wait_i)\\&\qquad Overhead_i = CatchUp_i - Wait_i;\\&\mathit{else}\\&\qquad Overhead_i = 0;\end{aligned}$$

  Clearly, no overhead exists if the wait time is longer than the catching-up time. As *keying* time (TPC 2002a) has not been used in the experiments the waiting between successive transactions consists of only *think times*. The think times had the negative exponential distribution as defined by TPC-C, scaled down so that the mean was an order of magnitude less than the values proposed in the standard.

Table 4-3 shows the results of *Experiment 1* (when the three replication schemes were subjected to the load of 20 TPC-C compliant clients). The values represent the mean transaction response times for each client. In addition, we include the results for experiments executed with individual servers as a reference. In this case the client application executed against the single servers without using the replica control features of DivRep. We can see that the replicated solutions are performing better than the single server configurations – no possibility for improving the *original* transaction response times (through early commencement of think times) exists with single server configurations. Different clients, of the same replication scheme, exhibit somewhat different response times, but the difference is insignificant (e.g. within ~5% for DivRep). This is partly due to the limited accuracy of the results available on the measurement machine in our testbed – the maximum resolution is 15 milliseconds (Microsoft 2004). We can see that DivRep is performing better than SimSI-Rep for this particular experiment type. DivRep, with both server configurations 1FB1PG and 2PG, demonstrates faster average transaction response times.

The results of DivRep with the pair of non-diverse servers, 2PG, are slightly better than the results of the diverse pair. This observation can be explained with the difference in CPU load of the two diverse servers under the particular workload. When DivRep employed the diverse pair, FB server was considerably more CPU bound than the PG server under *Experiment 1* and as a result the faster responses for the last DML operation in a transaction originate from PG server in most of the cases.

If we look at Table 4-4 we can see that the PG server is faster to execute all DML operations in a transaction in almost 68% of the times. While the situation is similar in

Experiment 2 (the percentage of times when the FB server was either faster than the PG or it finished the last DML at the same time as the PG marginally dropped), it changes in favour of FB in *Experiment 3* in which both servers become CPU bound and PG produces the fastest responses to the last DML in less than 50% of transactions. This observation explains why in *Experiment 3* DivRep with the diverse pair exhibits faster average response times for all clients than when DivRep is used with two PG servers. The average transaction response time, for all the clients executing against 1FB1PG, during *Experiment 3*, is approximately 200 ms while it is around 275 ms for 2PG configuration. This result shows that the use of diverse redundancy could bring performance improvements over non-diverse configurations.

Table 4-3 The mean values of user-view transaction response times for the three replication schemes (DivRep 1FB1PG, DivRep 2PG and SimSI-Rep 2PG) and single server configurations (1FB and 1PG), given as a reference, in the *Experiment 1*, when TPC-C workload was employed with 20 Clients.

Client	DivRep 1FB1PG AVG (ms)	DivRep 2PG AVG (ms)	SimSI-Rep 2PG AVG (ms)	1FB AVG (ms)	1PG AVG (ms)
1	200	192	352	936	728
2	200	184	352	944	720
3	200	184	352	928	712
4	200	192	352	936	712
5	200	184	352	936	720
6	200	192	352	936	720
7	200	184	352	936	720
8	200	192	352	936	720
9	200	184	352	936	712
10	200	192	352	920	712
11	192	184	368	928	704
12	192	184	368	936	720
13	200	184	360	936	712
14	200	184	368	936	704
15	200	192	368	936	712
16	192	192	368	928	720
17	200	184	368	936	720
18	192	184	368	936	720
19	200	192	360	928	712
20	200	192	368	944	720

Table 4-4 The percentage of times when a particular server (PG or FB) was faster to produce the result for the last DML in a transaction. Note that the percentage values in "FB faster" column include the occurrences when the logged times for the two servers were equal.

Exp. 1		Exp. 2		Exp. 3	
PG faster	FB faster*	PG faster	FB faster*	PG faster	FB faster *
68%	32%	72%	28%	48%	52%

Instead of using only summary statistics, such as mean and standard deviation, we analyse the performances more thoroughly using cumulative distribution functions

(CDFs). Figure 4-13 shows the CDFs of two random variables, *UV transaction response time* and *overhead*, for the first type of experiment (*Experiment 1*). We first examine the CDFs of UV transaction response times: T(DivRep 1FB1PG), T(DivRep 2PG) and T(SimSI-Rep 2PG). There is a stochastic ordering between SI-Rep and each of the two versions of DivRep, with either 1FB1PG or with 2PG. This confirms that DivRep performs better than SimSI-Rep. It also means that the use of diversity would be beneficial. On the one hand, the result is not surprising because it is likely that UV transaction response times are close to the corresponding *original* transaction response times (in which the notification of the successful commit, and not the result of the last DML, is regarded as the end time) when SimSI-Rep is used. In SimSI-Rep only one server executes the read operations of each transaction and only the writes are possible source of variability of the two servers - the end timestamps of UV transactions are likely to be close to the end timestamps of the corresponding original transactions. On the other hand, the load on the servers when SimSI-Rep is used is smaller than the load imposed by DivRep, and as a result one would expect to see shorter *original* transaction response times with SimSI-Rep so that UV transaction response times are reduced, too.

However there is no stochastic ordering between the distributions of the two different server combinations employed with DivRep. The ordering between the two distributions changes. This is particularly evident if we look at the corresponding CDFs between 45th and 60th percentile (T(DivRep 1FB1PG) and T(DivRep 2PG) in Figure 4-13). This indicates that the *pdf*s (probability density functions) of the two distributions have several cross-over points (Figure 4-14). The *pdf*s have a similar spread: this is not surprising since PG is the individually faster server and it produces the result for the last DML most of the times when the diverse pair is used. Nevertheless, the use of the diverse pair can help decrease variability - the probability density of 1FB1PG (represented by the green dashed line) has a smaller value of standard deviation (an aggregate measure of variance) and a shorter tail and, thus, it is favoured over the 2PG configuration. Besides, the usefulness of the latter configuration with DivRep is limited, since no benefits in terms of dependability assurance exist.

If we look at the overhead distributions (O(DivRep 1FB1PG), O(DivRep 2PG) and O(SimSI-Rep 2PG)) we observe that DivRep exhibits greater performance penalty, i.e. longer overhead times than SimSI-Rep. The overhead values are significant when

compared to the UV transaction response times (T(DivRep 1FB1PG), T(DivRep 2PG) and T(SimSI-Rep 2PG)) as the x-axis shows the logarithmic values. It is not surprising that SimSI-Rep exhibits the least performance penalty. On one hand SimSI-Rep has not got a potential to reduce the transaction times, by producing faster responses with both replicas; but, conversely, no overhead would be induced due to large *catching up* times of the slower server (more than 85% of all transactions incur zero-valued overhead).

Note that the jagged lines of the CDFs representing the UV transaction response times (Figure 4-13) are due to the time measurement quantisation. The horizontal steps are multiples of 15 ms, which is the length of the clock interval in Windows NT (Solomon and Russinovich 2000) (the machine on which the measurements were taken was running Windows 2000). The actual clock frequency of 65Hz (15.384615ms) explains the occasional tiny steps – 15ms vs. 16ms. The clock resolution puts the limit on the accuracy of the measurements and this is especially evident in short response times, e.g. experimental logs contained the following values for the response times less than 100ms: 15ms, 16ms, 31ms, 32ms, 46ms, 47ms, 62ms, 63ms, 78ms, 79ms, 93ms, 94ms. This limit is not obvious in the measurements of *overhead*s because the values are "smoothed" after subtracting wait times from the corresponding catching-up times.

The stochastic ordering of transaction response times between DivRep using the diverse pair of servers and SimSI-Rep with the marginally faster server can be observed in the other two experiments, when the read-oriented profile was used with 100 and 200 clients (Figure 4-15 and Figure 4-16 respectively). DivRep performs better for any level of confidence. Nevertheless, as in *Experiment 1* the overhead values are greater when DivRep is used. Hence the gain achieved with the shorter transaction times with DivRep is compromised with the long overhead times. An interesting question, then, becomes: is it possible to minimise the impact of the overhead? One possibility is to observe longer wait times (Figure 4-12), e.g. think times, which would reduce the overhead by masking the catching up times. Longer think times are realistic because we used 10-fold smaller values of the mean think times in the experiments than the ones specified in the TPC-C standard. We recalculated the CDFs of the overhead using the think times' values as specified in the TPC-C, applying 10-fold increase to the experimental think times recorded in the logs. Consequently, the probability that we observe a zero-valued overhead increased

from 10% to 40% for DivRep employing the diverse pair in *Experiment 3*. This calculation neglects the fact that the load on the servers would have changed if longer think times had been used. As a consequence, the overhead could have been further decreased. The same effect could have been achieved had we, in addition to think times, used other types of *wait* times, e.g. *keying* times, in our experiments.

At the same time, one would like to know why, especially under the higher loads (100 and 200 clients), the overhead times are so significant. In particular, is it the tardiness of the slower server, that can be approximated as *CatchUp - Commit* time (Figure 4-12), or the execution of the transaction boundaries, e.g. *Commit* time in Figure 4-12, which includes the waiting for *tb_mutex* (Section 3.1.1) acquisition, the main contributor to the duration of overheads? We answer this question with Figure 4-17, in which we plot the CDFs of the transaction boundary times (both *commits* and *begins*) and the *catching up* time for the experiment with DivRep employing the diverse pair (1FB1PG) under the highest load (*Experiment 3*). Clearly, the transaction boundary times are the main contributor to the overhead times and thus minimising their impact would benefit the performance of DivRep. This is the topic of the following section (Section 4.5).

To understand further the relation between UV transaction response times and the overheads we have calculated the correlation between the two variables (Table 4-4). There exists somewhat strong positive correlation between the two variables when DivRep is used with 2PGs, especially in the experiments with the read-intensive workload. This might be attributed to the fact that the individual servers become CPU-bound under these experiments. The values of the correlation for DivRep scheme employing the diverse pair, though less pronounced, are still significant in all experiments, while the correlation values are negligible for SimSI-Rep. The positive correlation for the experiments with DivRep exists, as it is likely that if the faster server takes long to execute all DMLs, i.e. long UV transaction response times are observed, the same will hold for the slower server, i.e. relatively long catching up times will be observed (particularly when the pair of the non-diverse servers is used). This is not true for SimSI-Rep because the two servers execute last DML in a transaction without (significant) time lag, the infrequent overheads are short and mainly due to the transaction boundaries, whose durations do not change as a result of variable database server processing.

We can observe a trend of increased correlation when DivRep is used under higher loads. This is because overhead values tend to be greater than zero under higher loads (an overhead will have a non-zero value when the catch-up time is greater than the corresponding think time). This is confirmed if we compare the magnitude of zero-valued overhead frequencies in Figure 4-13, Figure 4-15 and Figure 4-16 – the first figure depicts the highest frequency of zero-valued overheads. Thus, the correlation between the transaction times and the overheads is stronger. The somewhat significant correlation between the two variables is confirmed with the scatter plot in Figure 4-18 depicting the results of the experiment with the diverse pair. The data tends to move from the lower left to the upper right corner, indicating positive correlation to some extent. A similar picture has been observed in the other scatter plots representing DivRep; when 2PG configuration was used as well as under the other types of experiments. Most of the dots are placed above the unit slope indicating that the overhead values are greater than the corresponding UV transaction response times - the latter is relatively short compared to the durations of the catching-up times. The different shades of green depict the degree of overlap of particular pairs of values. The limited precision of the time measurement quantisation, described above, is evident in this figure too. This is the reason why, for example, the leftmost vertical band of values is thinner than the adjacent band, where two columns of dots are placed one next to each other.

With the user-centric analysis we have shown that the variability in the results of the diverse servers might be exploited for performance gain. Using *user-view* transaction response time as the measurement, we showed that the diverse pair performs better on average than the non-diverse pair of the marginally faster server deployed with SimSI-Rep, or DivRep under *Experiment 3*. Also, the variability of the results can be, to some extent, decreased using the diverse pair. However, the gain is achieved under the specific provisions and high *overhead* values, observed with the diverse pair, represent a considerable drawback. The impact of the overheads can be minimised if the longer wait times are used. This seems to be a realistic possibility since the think times used in the experiments were scaled down values of the ones specified in TPC-C standard. Additionally, the use of other wait times' types, e.g. keying times, would decrease the overhead further. The overheads are caused mainly by the serialisation of transaction boundaries and minimising the effect would be beneficial for the performance of DivRep.

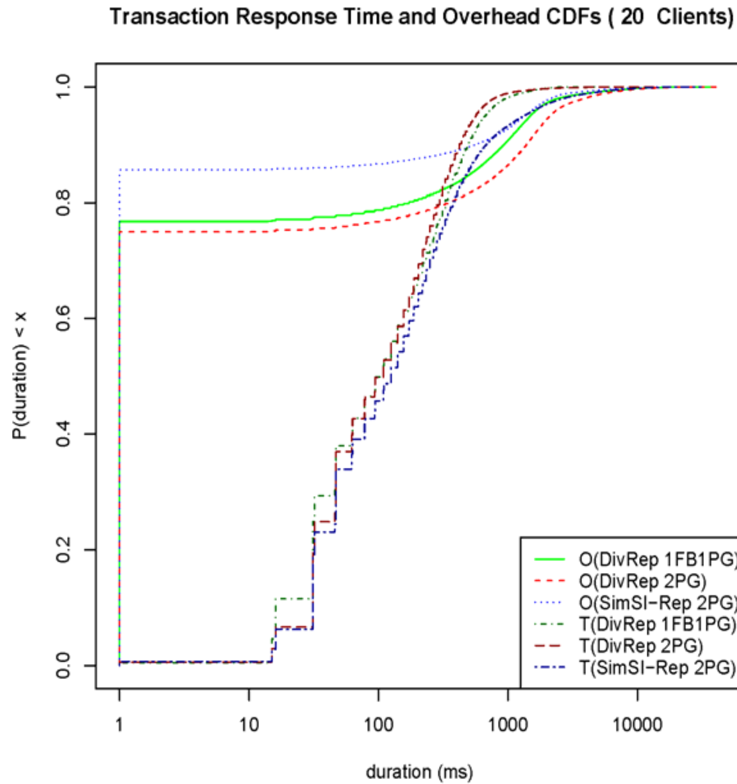

Figure 4-13 The experimental CDFs of user-view transaction response times (T) and the overhead (O), the performance delay incurred by the "catching up" of the slower server, calculated for *Experiment 1* for the three replication schemes, DivRep with 1FB1PG, DivRep with 2PG and SimSI-Rep with 2PG.

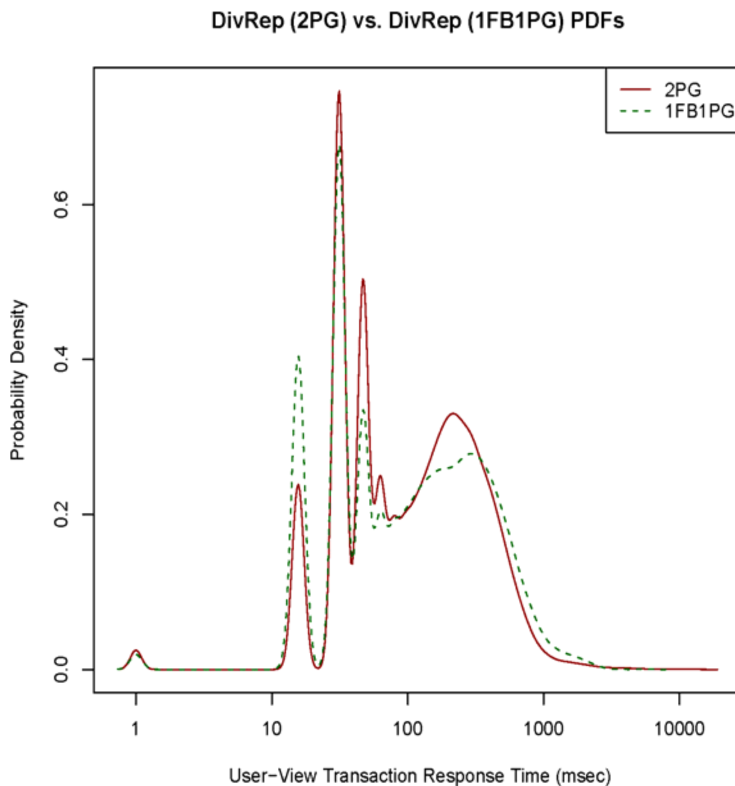

Figure 4-14 Probability density function (*pdf*) of UV transaction response times for two different server configurations (a non-diverse, 2PG, and a diverse, 1FB1PG) deployed with DivRep middleware under *Experiment 1*.

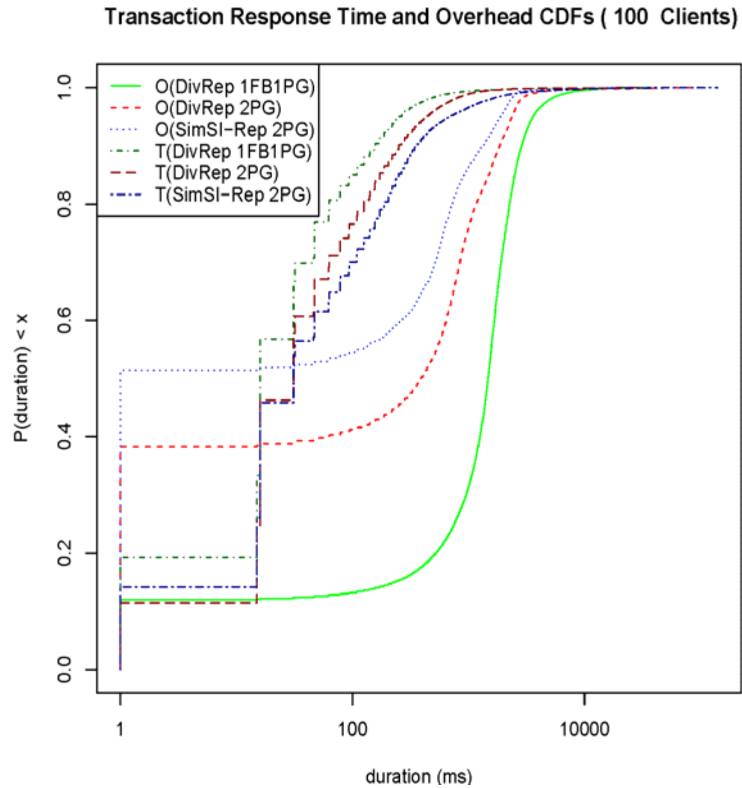

Figure 4-15 The experimental CDFs of *user-view* transaction response times (T) and the overhead (O), the performance delay incurred by the "catching up" of the slower server, calculated for *Experiment 2* for the three replication schemes, DivRep with 1FB1PG, DivRep with 2PG and SimSI-Rep with 2PG.

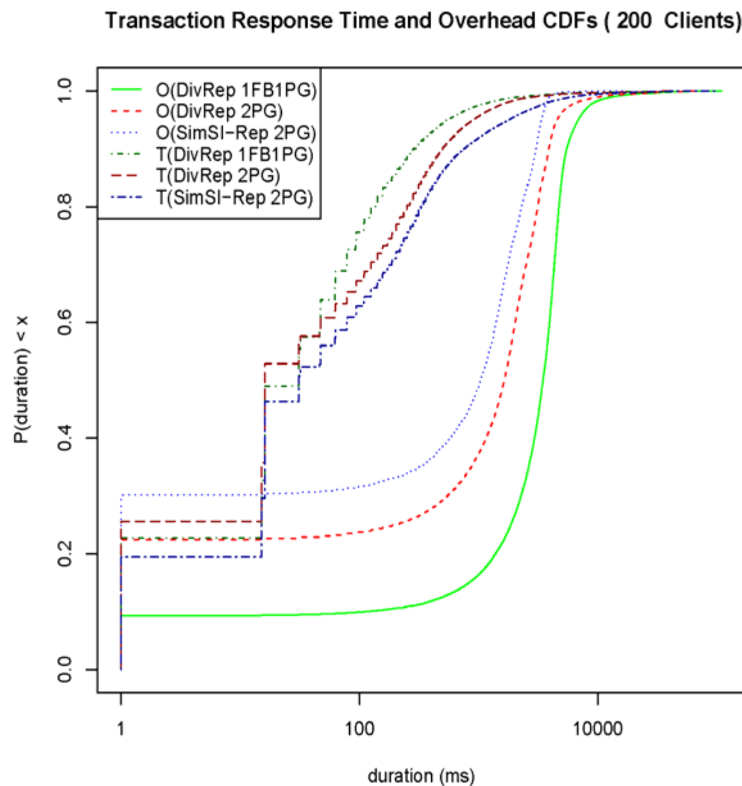

Figure 4-16 The experimental CDFs of *user-view* transaction response times (T) and the overhead (O), the performance delay incurred by the "catching up" of the slower server, calculated for *Experiment 3* for the three replication schemes, DivRep with 1FB1PG, DivRep with 2PG and SimSI-Rep with 2PG.

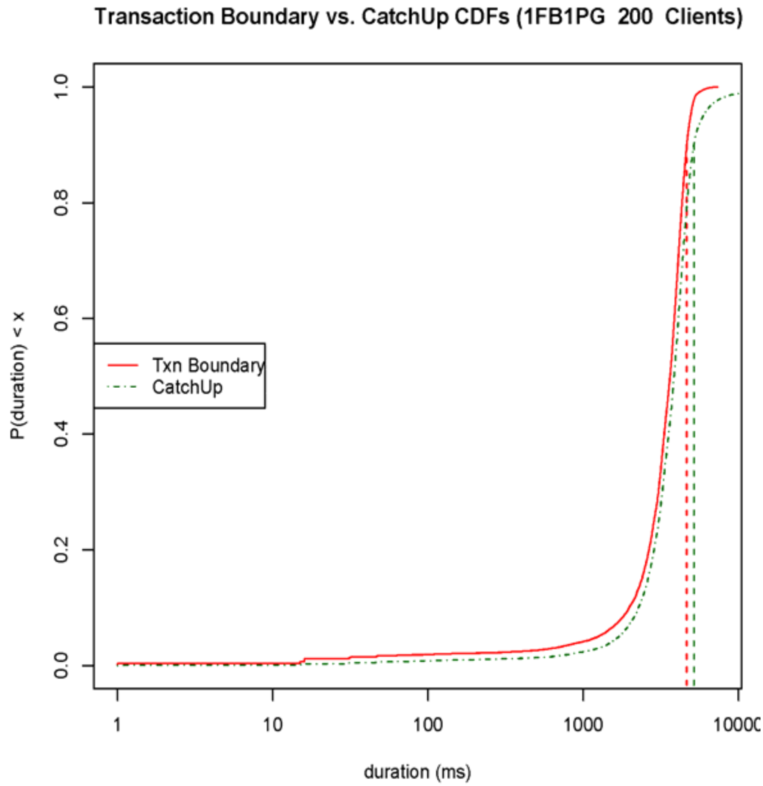

Figure 4-17 The experimental CDFs of the transaction boundary times and the catching-up times calculated for the diverse pair (1FB1PG) under *Experiment 3*.

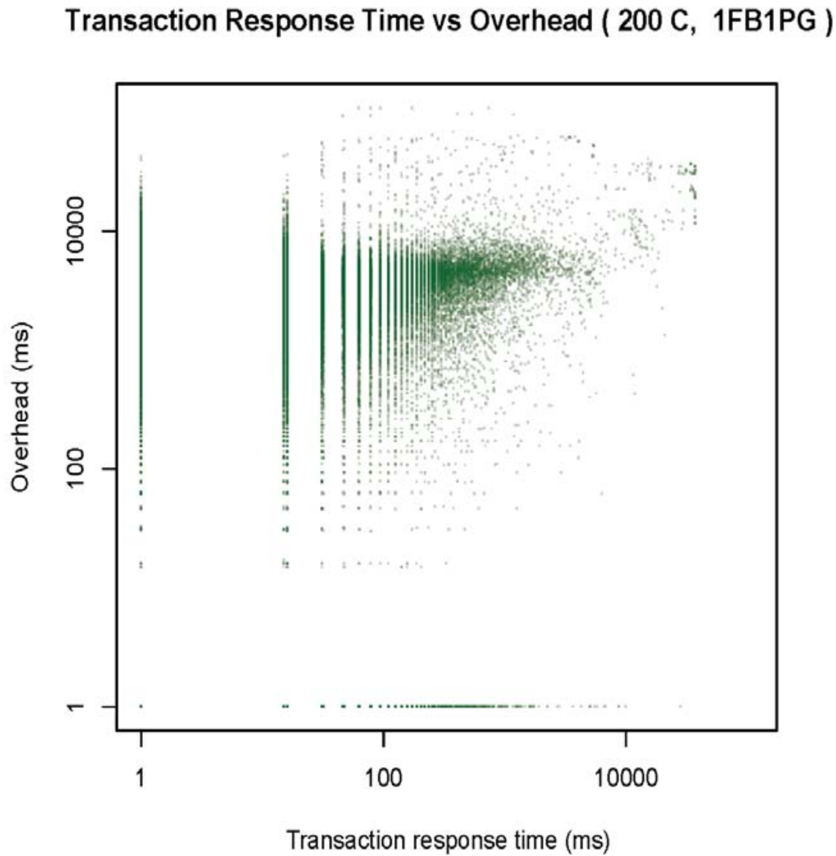

Figure 4-18 A scatter plot of *user-view* transaction response times and the corresponding *overhead* values for DivRep scheme using diverse pair (1FB1PG) under *Experiment 3*.

Table 4-5 Correlation coefficient values between *user-view* transaction response time and the *overhead* for the three replication schemes (DivRep with 1FB1PG, DivRep with 2PG and SimSI-Rep with 2PG) under the three experiments.

	Corr. Coefficient
TPC-C Profile (20 Clients)	
1FB1PG - DivRep	0.18
2PG - DivRep	0.14
2PG - SimSI-Rep	0.00
Read-intensive Profile (100 Clients)	
1FB1PG - DivRep	0.20
2PG - DivRep	0.57
2PG - SimSI-Rep	0.01
Read-intensive Profile (200 Clients)	
1FB1PG - DivRep	0.29
2PG - DivRep	0.56
2PG - SimSI-Rep	-0.02

4.5. Minimising Replication Overhead Using Priority Mechanisms

4.5.1. The Problem

Experimental evaluation presented in Section 4.4.4 revealed the following: imposing a serial order of transaction boundaries (BEGIN and COMMIT operations) on different replicas using DRA algorithm (Section 3.1.1) incurs performance overhead. Multiple transactions, initiated by different clients, might be attempting to simultaneously execute a transaction boundary. The middleware handles transaction boundary requests using a mutex, *tb_mutex*. No provisions for a particular boundary execution order are in place – it is the underlying implementation of the mutex that defines the execution order, e.g. in our implementation of DivRep the order is dictated by the underlying JVM (Java Virtual Machine). Only one transaction boundary is permitted to execute at a time, thus a client might be blocked by others, without the possibility to progress until it is granted the mutex. This *serialisation* of transaction boundaries introduces *lock convoy* effect, (Rinard and Diniz 2003), (Lampson and Redell 1980) and has negative impact on system performance. One strives to decrease or eliminate the effect of the performance problem.

Naturally, the number of simultaneously blocked clients depends on the concurrency degree. It, also, depends on the ratio between the duration of the transaction boundary operations, T_b, and the duration of the transaction's DML operations, T_{DML}. The larger the ratio between the two, T_b/T_{DML}, the greater the chance many clients will be blocked. If the ratio is small, i.e. the execution of the boundary operations is significantly shorter than the execution of DML operations, it is likely that many clients will be busy executing the DMLs and as a result the contention for the boundary operations will be smaller. If the durations of transaction boundaries are long relative to the DMLs, however, the chance that multiple clients wait for the mutex is greater. When the COMMIT operation is executed, the T_b duration depends on the transactional profile. If the transactional profile is write-intensive the COMMIT operations will be longer because the changes will have to be flushed to the disk i.e. there is an I/O overhead of writing out all pages affected by the transaction, such as data and index pages and similarly REDO/UNDO log has to be written. Correspondingly, long execution times of transaction BEGINs could be observed in

DivRep. This observation can be explained as follows. We use a dummy SELECT operation to start a transaction. The reason is that JDBC interface does not support explicit BEGIN operation but assumes a transaction starts upon the first operation after a COMMIT or an ABORT. In order to serialize transaction boundaries we introduced the dummy SELECT operation that reads a table from the database. Although the duration of the query is short on average, occasionally the data has to be fetched from the disk, at which times the execution duration significantly increases. This is the reason why in the cases when database does not reside fully in the main memory, and cache *hit ratio* is poor, an expensive I/O operation has to be initiated.

The replication algorithm of DivRep middleware introduces an overhead due to the serialisation of transaction boundaries. We have taken detailed measurements to enable us to accurately evaluate the impact of this serialisation. In particular we recorded the following measures of interest:

- Transaction response times, T_t. It is measured as the time between a client sends the BEGIN command to start a transaction until it receives the notification that the COMMIT has been successfully executed (after the middleware have executed 2PC-DR (Section 3.1.1)) so that it can start the following transaction. Clearly, this time includes the execution of the transaction boundaries and the DMLs on both servers.
- *Client-view* transaction boundary response time, T_b^C . In the rest of the document T_b^C will be used to refer to response time of either BEGIN or COMMIT commands if not explicitly specified otherwise.
- *Server-view* transaction boundary response time, T_b^S.

We make a distinction between client-view and server-view boundary response times (Figure 4-19): T_b^S captures the execution of the actual SQL operation (BEGIN or COMMIT) on a DBMS, while T_b^C includes the waiting time of each transaction to acquire *tb_mutex* too, hence $T_b^C \approx T_b^S + T_{WAIT}$, where T_{WAIT} represents the waiting time for *tb_mutex* acquisition. T_{WAIT} includes time spent by the DBMSs on execution of DML operations from concurrent transactions, since both types of SQLs (DML and transaction boundary operations) compete equally for the resources, i.e. it is possible that execution of the DMLs blocks the concurrent transaction boundary operations. As the number of clients increases, the difference between the two types of boundary response times becomes greater. T_{WAIT} increases because, under high load, the contention is greater and the frequency of transaction boundary operations is higher.

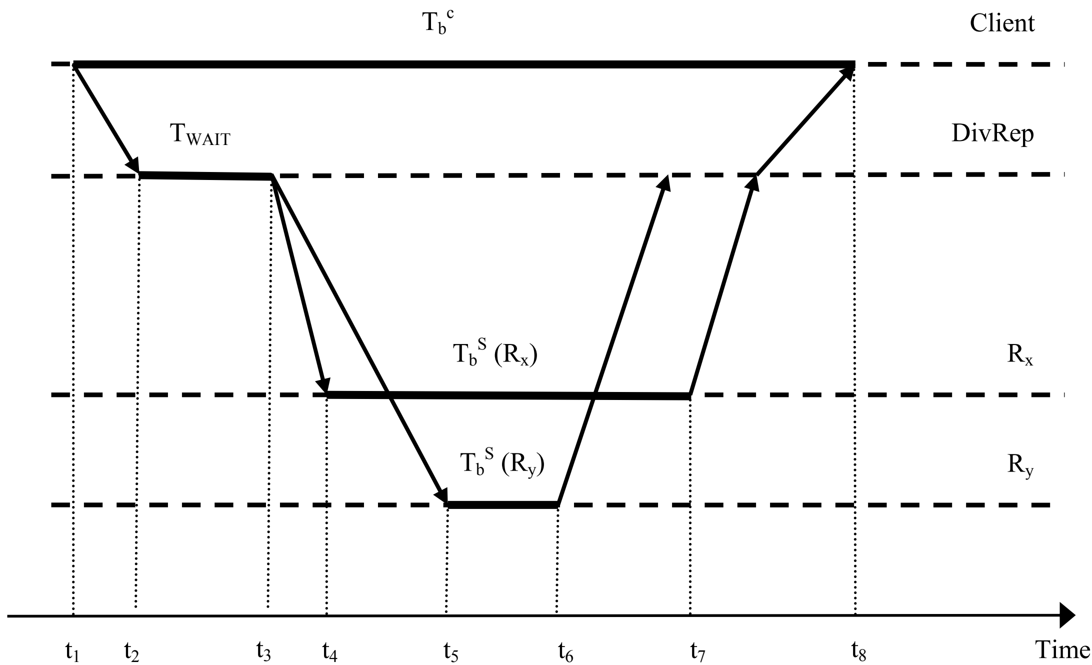

Figure 4-19 Transaction boundary duration as perceived by different parts of a replicated system: *Client-view* transaction boundary response time, T_b^C, calculated as $t_8 - t_1$, and *server-view* transaction boundary response times, T_b^S (R_x), calculated as $t_7 - t_4$, and T_b^S (R_y), calculated as $t_6 - t_5$. The difference between the T_b^C and a T_b^S might be significant due to the T_{WAIT}, calculated as $t_3 - t_2$, time needed to acquire the shared mutex. Please note that the execution of a transaction boundary on two replicas might not overlap in real time, one of the servers might finish the execution before the other one starts it.

We experimented with a replicated server configuration when two FB servers are deployed. The reason why we used FB servers is that the implementation of a specific solution (Section 4.5.2) we offer for minimising the serialisation overhead requires a substantial change of PG's functionality – the server processes should be runnable by privileged (*root*) user (this feature is unavailable in PG by default). However the results obtained with a pair of FB servers would apply to any replicated setup. The solution does not depend on any specifics of FB server.

The choice of hardware was the same as in the experiments described in Section 4.4. The client application was executing the write-intensive profile specified by TPC-C standard. The database size was three times bigger than the available RAM. We varied the number of clients to evaluate the impact of load on the serialization overhead. We executed experiments with 20 and 50 clients.

Figure 4-20 shows the response times of *client-view* BEGIN operations (T_b^C BEGIN) plotted against corresponding transaction response times, T_t. A significant portion of the transaction response times is comprised of the corresponding client-view BEGIN operations. This is not surprising since transaction latency includes the potentially

long T_{WAIT} times. The average response time of a BEGIN operation is almost exactly one fifth of the average transaction response time. The figure shows, however, the variability of both measures. Similar results were obtained for the COMMIT operation too. During the experiments we measured the CPU utilisation on the database server machines. We established that 25%-30% of the CPU resource was not used – it was reported by the Linux resource consumption utilities as idle. Therefore we could not achieve the maximum performance with the hardware used in the experiments. Although the underutilisation of CPU can be explained with noticeable I/O activity, it was clear that the serialisation of boundary operations has contributed to the performance bottleneck, too. This was confirmed with greater performance penalty once we increased the concurrency degree.

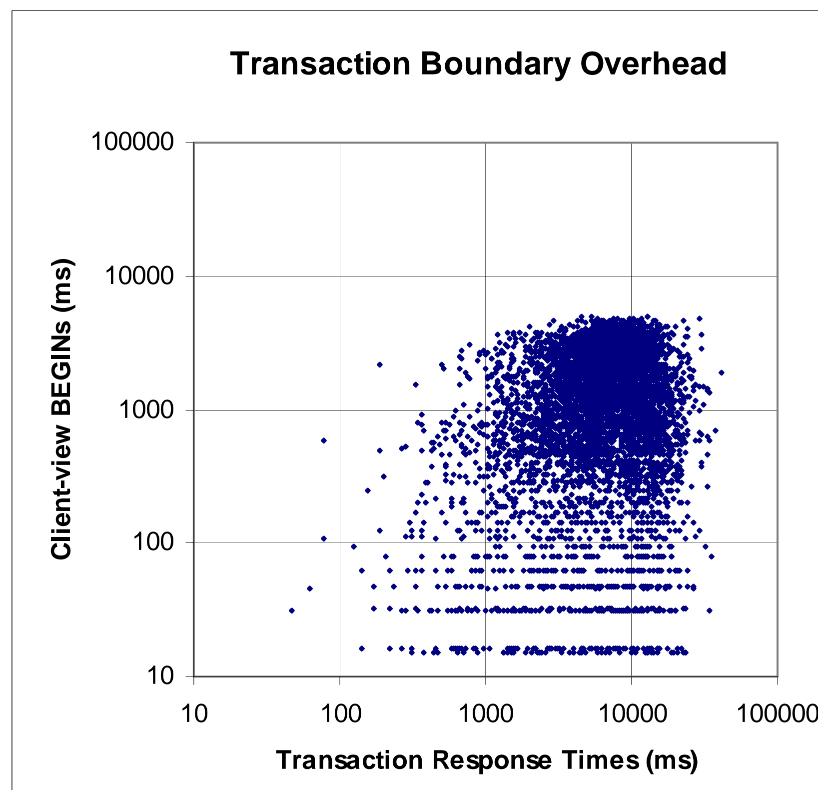

Figure 4-20 Transaction response times and the corresponding client-view BEGIN response times for the experiment with 50 Clients.

4.5.2. The Solution

In order to reduce the overhead of serialising transaction boundary operations we introduced a prioritisation mechanism to improve their performance. We achieve this using a particular process priority policy as follows. Prior to the execution of a transaction boundary, we would programmatically increase the CPU priority of the corresponding server process on each replica. A complementary operation to restore

the default process priority would be called once the execution of transaction boundary finishes. In this way boundary operations are executed using higher priority values, while DML operations execute with lower priorities. The prioritisation of transaction boundaries reduces the time a transaction waits to get hold of *tb_mutex*, T_{WAIT} time – contention for DBMS resources between transaction boundaries and DMLs of concurrent transactions is reduced, since the servers do not schedule the latter ones as long as there are boundary operations to be executed. Thus, the client will observe shorter latency of transaction boundary operations, i.e. T_b^C will decrease. Consequently we have implemented a User Defined Function (UDF) on each database replica, referred to as `setProcCPUPrio`. The UDF invokes a kernel API function, *`setpriority`*, for raising CPU priority of a particular database process. The UDF has been implemented using C programming language and its source code is as follows:

```
#include <sys/time.h>
#include <sys/resource.h>
int setProcCPUPrio(int* nice)
{
      return setpriority(PRIO_PROCESS,0,nice);
};
```

The PRIO_PROCESS parameter value in the invocation of the *setpriority* function specifies that the priority of a process should be modified (alternatively, one can change the priorities of a group of processes or the priorities of all processes belonging to a specific user, by specifying PRIO_PGRP or PRIO_USER, respectively). The value of zero specified for the second parameter in the invocation of *setpriority* indicates that the priority of the current process (the one that invokes the UDF) should be changed. Hence when the UDF is invoked from a database connection, using the standard JDBC interface (Figure 4-1), the priority of the process serving the connection will be changed. The value of the *nice* parameter indicates, by manipulating the entries in the kernel's scheduler, with which priority the process should execute. Commonly, the values of the *nice* parameter range between -20, which signifies the process of the highest priority, to +19, which represents the lowest priority process. The default value of 0 is usually inherited from the parent process. The priorities determine the *time quantum* of a process – the higher the priority (i.e.

the lower the numerical value of the *nice* parameter) the longer the time quantum (Bovet and Cesati 2005).

Prioritisation of the server processes on its own is insufficient for an effective performance improvement of response times of transaction boundary operations. This is because a corresponding prioritisation has to be implemented in the middleware too. Therefore we have modified the priority of each middleware thread communicating to a particular database replica: prior to execution of transaction boundaries, each thread's priority was increased and upon the end of the execution it was restored to the default value (recall from Section 4.1 that the middleware is implemented as a multithreaded application – each client is served with *n* number of threads, where *n* is the number of deployed replicas).

As mentioned above, the execution of transaction boundaries might be an expensive I/O operation. Hence, one might wonder why, beside the CPU prioritisation policy, we do not manipulate the I/O priorities, too. The reason is that we do not attempt to decrease the actual time spent by the DBMSs to execute the transaction boundary operations (T_b^S time), by giving it a higher I/O priority. It is the time "wasted" for the acquisition of *tb_mutex*, T_{WAIT} time, which we want to minimise, by eliminating the possibility that the execution of DMLs contend with concurrent boundary executions for the CPU resources. Let us aid the understanding of the idea with Figure 4-21. Two database processes, p_1 and p_2, execute a COMMIT operation and a DML operation respectively, on a CPU. Panel a) shows an interleaved execution of the two operations causing process switches, and corresponding scheduling, to occur due to the assigned time quantum being longer than the respective durations (*reschedule 1* and *reschedule 2*) or due to operation termination (*reschedule 3*). In this way, the execution of both operations is performed in two parts. If the time quantum of the process p_1 had been made longer, by increasing the respective *nice* value, the COMMIT operation would have terminated during its first epoch (panel b)). The subsequent process switch (*reschedule 4*) would then assign the CPU time to p_2. This would decrease the latency of the transaction boundary operation and shorter T_{WAIT} time would be observed. The figure shows only a special case when two processes are contending for the CPU time. When the same priority is applied to all processes, under the higher load the overhead would be bigger – the execution of transaction boundaries would be interleaved with multiple DML operations.

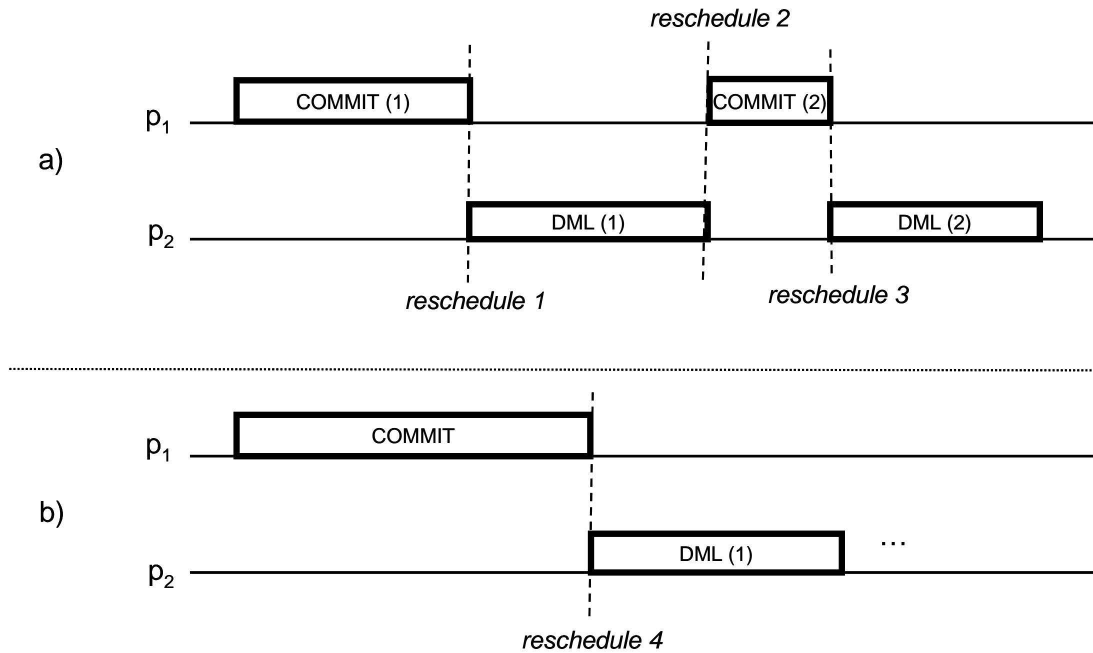

Figure 4-21 An interleaved execution of (parts of) two processes on a CPU.

Table 4-6 shows the average response times and the corresponding standard deviations for the two types of experiments under different loads of 20 and 50 clients:

- The results of the baseline experiment without the process prioritisation – *Same Prio*.
- The results of the experiment with the variable process priorities – *Different Prio*.

The decrease in the average response times of transaction boundaries is evident. It is reduced from approximately 50% for COMMITs to around 87% for BEGINs under the load of 50 clients. In the experiment with variable process priorities the CPU utilisation went up to 100%, i.e. the hardware resource was now fully utilised.

Although we can observe that average transaction response time decreases with variable process priorities too, the performance improvement is not pronounced as with transaction boundary operations. The relative improvement is 12% under the load of 20 clients and 6% under the load of 50 clients. As a consequence the experiment duration is shorter too. One of the factors that negate the performance improvement of transaction boundaries is a new type of overhead we introduced, namely the execution of the UDFs that *promote* and *demote* server processes, which inevitably has consumed part of the underutilised CPU resource. More importantly, while we successfully improve the performance of one part of transaction, T_b, we inevitably slow down its other part, T_{DML}. This behaviour of penalising low priority processes is a characteristic of multitasking on a *uniprocessor* machine. An interesting

observation is that the performance of transaction boundaries become more predictable (compare the corresponding values of the standard deviation in Table 4-6). Though to a lesser extent, the same is true for the transaction response time under the load of 20 clients. Under the higher load, however, the standard deviation of transaction response time increases once the manipulation of the process priorities is introduced. The result would be a bad news for someone who seeks real-time performance (the worst-case behaviour deteriorates), but it is beneficial for conventional database systems that try to maximize average-case performance and resource utilization.

Table 4-6 Experiment duration, average transaction response times, average response times of *client-view* transaction boundaries and the respective standard deviations for the experiments with *same* and *different* server process priorities. Apart from experiment duration, all values are given in milliseconds.

Exp. Type	*Measure of interest*	*20 Clients*		*50 Clients*	
		AVG. (ms)	*ST. DEV. (ms)*	*AVG. (ms)*	*ST. DEV. (ms)*
Same Prio.	Exp. Duration	16 min.	n/a	17 min.	n/a
	Tran. Time	3959	2776	7550	4948
	BEGIN	357	515	1500	1172
	COMMIT	1788	1614	3879	2954
Diff. Prio.	Exp. Duration	14 min.	n/a	15 min.	n/a
	Tran. Time	3472	2741	7123	6219
	BEGIN	70	149	183	370
	COMMIT	852	1009	2026	2089

4.5.3. Discussion

The experimental results presented here show convincingly that the characteristics of the replication algorithm can be improved using priorities. At first it may seem paradoxical that adding more work to the database server machines (controlling the priorities of the server processes serving the connections) makes the server run faster. There is, however, nothing magical in this.

Without any explicit measures to reduce the duration of transaction boundaries serialisation may become a *bottleneck* in the replication scheme. If this is the case the concurrent database connections cannot run at a maximum pace. Instead, the pace is limited by the delays on transaction boundaries. Increasing the number of concurrent

transactions will make this problem more and more acute – the transactions will wait longer and longer to enter a critical section to set transaction boundaries atomically across the replicas.

This problem is not specific to our implementation. It would be observed whenever clients may block each other if a scheduling mechanism of their requests operates (e.g. at the middleware level) without control on how the CPU is allocated by the DBMS. For example, imagine a replication scheme avoids conflicts between writing transactions as follows. Whenever transactions access the same table for an update, all but one are allowed to progress while the others are delayed until all previous transactions which access the same table are done with their updates. This “serialisation” itself is a bottleneck for the writing transactions. Then because the DBMS resources are shared between concurrent writing and reading transactions, the cost of the serialisation is also affected by the reading transactions. Had a prioritization policy to favour writing transactions been in place, the cost due to the reading transactions could be reduced.

A straightforward way to decrease the duration of BEGINs on the database servers (reduction of the corresponding *server-view*, T_b^S, times), would be to execute a less costly operation (e.g. a query that does not read any table or a call to an empty function) instead of the dummy SELECT. However our experiments without the use of the process prioritisation (*Same Prio.* experiments) revealed that the current implementation of the BEGIN operations resulted in durations significantly shorter than the corresponding *client-view* durations. This is not surprising because DBMSs schedule DMLs as likely as concurrent BEGINs, and thus client might observe long transaction boundaries delayed by the execution of several DMLs.

Once a bottleneck at transaction boundaries has occurred then the database servers may receive a workload (amount of DMLs from the connections, which have passed the serialisation bottleneck) lower than what their CPU(s) can handle. The particular workload passed to the database servers is clearly dependent, as discussed in Section 4.5.1, on the level of concurrency and the particular types of transactions. With the same level of concurrency (number of concurrent clients), the serialisation of boundary operations will be more of a problem for ‘short’ transactions than for ‘long’ ones. Another dimension, which will affect whether the boundary serialisation may become a bottleneck or not, is how powerful the CPU(s) is (are) on the database servers. Given a particular type of transactions and a level of concurrency, the CPU

underutilisation will be worse with more powerful machines than with less powerful ones.

Whenever CPU resource is underutilised without priorities (i.e. the boundaries serialisation has become a bottleneck), using priorities may help. The actual gain, of course, will depend on how much of the CPU spare resource without priorities will be consumed on manipulating the priorities of the processes on the database server machine(s). The higher the underutilisation without priorities the better chance one has to improve the server throughput by introducing priorities.

Clearly, if there is no CPU underutilisation without priorities, introducing priorities will only make things worse. After we add more work for the database servers to do (manipulate their process priorities) the performance will deteriorate, since the servers would have been already overwhelmed by the workload generated by the clients.

In summary, the scheme that we presented is not guaranteed to always lead to performance gains. Fortunately, in cases important in practice of very powerful database servers (e.g. the ones using Symmetric Multiprocessing (SMP)), and short transactions, the scheme is likely to deliver performance gains.

The particular implementation of the prioritisation scheme is tailored for the database servers with the process-based architecture, where a dedicated server process serves every client connection. It would be, however, straightforward to implement the priority policy for thread-based architectures in which database server runs as a single process and client connections are served by multiple threads.

It is possible to use an alternative prioritisation policy in order to improve the performance of the replication algorithm. As detailed in Section 4.4.4 DivRep exhibits a performance overhead as part of the “catching-up” phase, due to the use of synchronous database replication scheme and an atomic commitment protocol. In addition to promoting priorities of database processes while executing transaction boundaries, the promotion could be applied to the slower server process, while executing in the “catching-up” phase, too. The policy would be applied as follows. Once the faster server executes the last DML operation of a transaction and the client has sent the commit request to the middleware, the priority of the slower server is promoted. The slower server continues the execution of the remaining SQL operations on the promoted process priority. Once the slower server has finished all the SQL operations in the transaction, the process priority of the faster server is increased and both servers commit the transaction. The process priorities are restored to the default

values only once the BEGIN operations of the following transaction is executed. In this way we avoid performing the manipulation of server process priorities twice: once the slower server enters the "catching-up" phase and the second time for the transaction boundary operations.

Undoubtedly, the idea of giving priority to operations in the critical path has been researched and applied in other contexts (some of which we describe in Section 4.5.4), but, to the best of the author's knowledge, it has not been used to minimise overhead incurred by database replication protocols.

4.5.4. Related Work

Similarly to database replication research, there is a vast literature on real-time database management systems (RTDBMS) and database transaction scheduling. Traditionally, these topics have had an impact on telecommunications, manufacturing and avionics industry where conventional databases could not deliver the satisfactory real-time performance. The early work of Abbott and Garcia-Molina (Abbott and Garcia-Molina 1992) developed algorithms for scheduling real-time transactions that operate on RAM-resident as well as disk-resident databases. The work of (Carey, Jauhari et al. 1989) proposes specific priority-based schemes for managing resources of database servers. The authors advocate that a buffer management policy has to complement any CPU and disk scheduling used for minimising a resource bottleneck. A more recent work (Ailamaki, McWherter et al. 2004) describes several non-preemptive and preemptive prioritisation policies for database servers. It shows that a few-fold improvement for high-priority transactions is possible using simple scheduling policies, without imposing a significant overhead on the low-priority transactions. The work of (Stankovic, Son et al. 1999) indicates the frequent misconceptions about the real-time requirements for databases, such as that the current database technology can solve real-time problems or that a real-time database must reside in the main memory. The research of (Hall and Bonnet 2005) show that a prioritized asynchronous I/O could help Linux-based database servers to fully utilize available I/O bandwidth using an aggressive I/O submission policy.

As a part of parallel and distributed computing research, different solutions to synchronisation overhead have been proposed. One such solution (Rinard and Diniz 2003) proposes adaptive replication technique that automatically eliminates synchronization bottlenecks in multithreaded programs.

5. Uncertainty-Explicit Assessment of DivRep Components

The quest for certainty blocks the search for meaning. Uncertainty is the very condition to impel man to unfold his powers.

Erich Fromm

In Section 4 we reported on performance implications of DivRep middleware using diverse database servers. One of the goals of the described studies was to measure the amount of 'diversity' that exists between different servers, e.g. a particular server might produce a fast response for a request the other one is slow on. While measuring diversity was the topic of the previous section, here we concentrate on selection of the servers, to be included in an FT-node.

We have observed in the previous section (Section 4) that the performance of DivRep middleware is highly influenced by the choice of the particular servers. In the experiments described in Section 4.4 a particular server (PG) is universally faster, for the chosen workload and load settings, than its counterpart in the FT-node deployment (FB). In this way the usefulness of deploying diverse database servers for performance improvement is limited. We pointed out that the issue could be alleviated with the use of another server, which performs considerably better than FB. Consequently, the performance of the FT-node, deploying two diverse servers with similar performance, could be improved.

One of the main goals of DivRep middleware, however, is to improve fault-tolerance of replicated database systems. In order to fulfil the requirement, DivRep middleware has to employ database servers with satisfactory level of dependability attributes. Yet we would like to keep the performance penalty as low as possible. Looking at the single attribute (e.g. PG server is universally faster than FB server), thus, is insufficient. Similarly, looking at dependability assurance alone is also insufficient. One would have to take into account both attributes when ranking the available servers before deciding which to use in DivRep.

The assessment of the attributes of DivRep components can follow the usual practice found in component-based software development, where assessment techniques crucially depend on assuming that the values of the assessed attributes will be known with *certainty*, at the end of the assessment (Kontio, Chen et al. 1995), (Jeanrenaud and Romanazzi 1994), (Ncube and Maiden 1999). However, since the assessment is carried out under various assumptions, which may not hold true in real operation, and with limited resources of time and budget it is clear that the outcome is subject to *uncertainty*. The assessors may never be 100% sure that what they concluded during the assessment will be confirmed when the component is used in operation. This is clearly true for some parameters, which can be estimated *objectively*, e.g. failure rate, performance, etc. For failure rate, for instance, even after a very thorough testing one can only identify a range of rates which are more likely than others. For instance, Littlewood and Wright have shown (Littlewood and Wright 1997) that starting with indifference between the values of the failure rate (i.e. uniform distribution of the failure rate in the range [0, 1]) and seeing a protection system process correctly 4600 demands translates into 99% confidence that this system's probability of failure on demand (*pfd*) is no worse than 10^{-3}.

In what follows we propose an assessment method in which the assessment results are subject to explicitly stated uncertainty and discuss how this may impact the decisions about the use of different servers. The method also enables representing the dependencies that exist between the uncertainties associated with the values of the component attributes which affect the decision about which of the available servers to choose and also encourages assessing the dependent attributes simultaneously, thus speeding up the assessment. We provide empirical results from a study with database servers, which demonstrate how the assessment method can be used in practice. Although the method has wide applicability, our main aim is to show that it could be used for selecting components to be included in a fault-tolerant node employing DivRep.

5.1. Motivation for Using Uncertainty-Explicit Assessment

The value of expressing the assessment results in the form (value, confidence) has been recognized in some other technical areas which dealt with assessment. The best performing software reliability-growth models which predict the failure rate from the

observed failures in the past, for instance, are those in which the model parameters are treated as *random variables* (Brocklehurst, Chan et al.). In these models the 'true' values of the attributes being assessed are never assumed known with certainty. Instead the attribute is characterized by a probability distribution from which the true value of the attributes will come (i.e. are seen as drawn at random). For each reliability target, then, the assessor can tell the probability that the true reliability is lower than the target. Such models *systematically outperformed* the alternative simplistic methods in which the parameters were assumed known with certainty (Lyu 1996). If the 'uncertainty explicit' models have been the best choice with one specific method of assessment – software reliability – it seems natural to try similar 'uncertainty explicit' methods for other assessments, e.g. selecting the best database server, from a set of comparable alternatives, for building the FT-node by evaluating their respective attributes.

There are various methods for representing uncertainty (Wright and Cai 1994). Bayesian approach to probabilistic modelling is one of the best-known ones and used with some success in reliability assessment (Littlewood and Wright 1997), (Lyu 1996). It allows one to combine, in a mathematically sound way, the prior belief (which may be 'subjective' and possibly inaccurate) about the values of a parameter or a set of parameters to be assessed with the ('objective') evidence from seeing the modelled artefact in operation. Combining the prior belief and the evidence from the observations in a mathematically correct way leads to a posterior belief about the values of the assessed attribute(s).

When selecting components for DivRep middleware we are particularly interested in performance and reliability of the database servers being compared. The selection of a particular database server is based on uncertain values of the attributes and as such should take into account a possible dependence between them. Ignoring the possible dependence between the attributes represents a particular form of belief: that assessing one of the attributes one can learn *nothing* about the other one. This form of belief might be justified in some cases, e.g. performance of a database server will hardly tell anything about the quality of its documentation and vice versa. The same belief seems ungrounded, however, in the case of assessing performance (e.g. response time) and reliability (e.g. failure rate). We may assume the opposite: that the uncertainties associated with these two attributes are independent, in the statistical sense. Under this assumption learning something about reliability will tell us nothing about

performance and vice versa. Now, suppose that we have run a very long testing campaign and have repeatedly observed that whenever the response was late it was also incorrect and no other incorrect response has been observed. With such evidence of a strong positive correlation between the failures (incorrect responses) and the responses being late, we may accept that any change of our belief about the rate of failure should also be translated into a change in our belief about the rate of late responses. Therefore, it is reasonable to use the concept in which the uncertainties associated with the assessed attributes may be *dependent.*

5.2. Bayesian Approach to Assessment of a Single Attribute

In this subsection we briefly summarize how the Bayesian approach to assessment is normally applied to assessment of a single attribute. Assume that the attribute of interest is the probability of failure on demand (*pfd*) of a DBMS. If the system is treated as a black box, i.e. we can only distinguish between DBMS's failures or successes (Figure 5-1), the Bayesian assessment proceeds as follows.

Figure 5-1. Black-box model of a DBMS. The internal structure of the component is unknown. Only its output (success or failure) is recorded on each demand and used in the inference of DBMS's *pfd*.

Let us denote the system *pfd* as p, with prior distribution (probability density function, *pdf*) $f_p(\bullet)$, which characterises the assessor's knowledge about the DBMS's *pfd* *prior* to observing the server in operation. Assume further that the DBMS is subjected to n demands, independently drawn from a 'realistic' operational environment (i.e. an operational profile, which can be defined as a quantitative characterization of how the component will be used in its 'true' environment (Musa 1993)), and r failures are observed. The posterior distribution, $f_p(x \mid r,n)$, of p after the observations will be:

$$f_p(x \mid r,n) \propto L(n,r \mid x) f_p(x) , \qquad \text{Eq. 5-1}$$

where $L(n,r \mid x)$ is the *likelihood* of observing r failures in n demands *if* the *pfd* were exactly x, which in this case of independent demands is given by the *binomial* distribution, $L(n,r \mid x) = \binom{n}{r} x^r (1-x)^{n-r}$. For any prior and any observation (r, n) the posterior can be calculated for any of the DBMSs included in the assessment. To be

precise, the posterior can be calculated either by using a conjugate prior distribution (Dickey 1982), in which case the posterior distribution is guaranteed to be in the same family as that of the prior for a given likelihood function (e.g. Beta distribution prior, with Binomial Likelihood function, gives us a Beta distributed posterior) or it can be calculated through numerical methods and approximations. In our case, since the conjugate family has limitations (Littlewood, Popov et al. 2000) we have used numerical methods to calculate the posterior.

Even if no failure is observed (i.e. $r = 0$), the posterior can be calculated. Other measures of interest can also be derived from this posterior, e.g. the probability that the DBMS will survive the next 5000 randomly chosen demands. This probability can be calculated for each of the DBMSs included in the assessment as follows:

$$\int_0^{\infty} (1-p)^{5000} f_p(p \mid r, n) dp$$

Then the best DBMS will be the one, for which the integral above gets a maximum value.

5.3. *A Model for Assessment of 2 Non-Independent Attributes*

In the selection process of optimal components we assess two non-functional attributes, DBMS's *pfd* and performance, the latter assessed in the form of whether a response is received on time or not, i.e. the probability of a *late response* on a demand, *pld*. In terms of comparison of several DBMSs using a binary score – on time vs. late – seems adequate. Any DBMS, which responds with an acceptable delay, might be regarded equally good from the point of view of the system's integrator.

Here we define a model to help with the comparison of DBMSs assessed by subjecting them to a *series of independently selected demands*. Both, the *pfd* and the *pld* of DBMSs, are used in the comparison and different comparison criteria are discussed.

On each demand the response received from a DBMS is evaluated from two different viewpoints: correct/incorrect and on time/late. Clearly 4 combinations exist, which can be observed on a randomly chosen demand, as shown in Table 5-1. The four probabilities given in the last column sum to 1. So if the last three probabilities are 0.2, 0.4 and 0.3, respectively, then the first one p_{10} = 1 - (0.2 + 0.4 + 0.3) = 0.1. This constraint remains even if we treat the probabilities in Table 5-1 as random variables:

their sum will always be 1. Thus, the joint distribution of any three of these probabilities, e.g. $f_{p_{01},p_{10},p_{11}}(\bullet,\bullet,\bullet)$, gives an exhaustive description of the DBMS's behaviour. In statistical terms, the model of the DBMS with two binary attributes has three degrees of freedom.

Table 5-1 The outcomes, their frequencies and probabilities for a random demand.

Event	Correct Response (Reliability)	Response On-Time (Performance)	Number of observations in *n* demands	Probability
α	No	Yes	r_1	p_{10}
β	Yes	No	r_2	p_{01}
χ	No	No	r_3	p_{11}
δ	Yes	Yes	r_4	p_{00}

The marginal probabilities of getting an incorrect response on a random demand, let's denote it p_I, and of getting the response late, p_L, respectively, can be expressed as:

$p_I = p_{10} + p_{11}$ and $p_L = p_{01} + p_{11}$.

p_{11} represents the probability of receiving late an incorrect response and, thus, a notation $p_{IL} \equiv p_{11}$ will capture better the intuitive meaning of the event it is assigned to. Instead of using $f_{p_{10},p_{01},p_{11}}(\bullet,\bullet,\bullet)$ another distribution, which can be derived from it through a functional transformation (*Change of variables* is a standard transformation method in *Calculus*, which requires the calculation of the Jacobian determinant), can be used. In this section we use $f_{p_I,p_L,p_{IL}}(\bullet,\bullet,\bullet)$, which is given by:

$$f_{p_I,p_L,p_{IL}}(\bullet,\bullet,\bullet) = f_{p_I,p_L}(\bullet,\bullet) f_{p_{IL}}(\bullet \mid p_I, p_L) \qquad \text{Eq. 5-2}$$

It can be shown that for a given observation (r_1, r_2, and r_3 in N demands) the posterior joint distribution can be calculated as:

$$f_{p_I,p_L,p_{IL}}(x,y,z \mid N, r_1, r_2, r_3) = \frac{f_{p_I,p_L,p_{IL}}(x,y,z) L(N, r_1, r_2, r_3 \mid p_I, p_L, p_{IL})}{\iiint\limits_{p_I,p_L,p_{IL}} f_{p_I,p_L,p_{IL}}(x,y,z) L(N, r_1, r_2, r_3 \mid p_I, p_L, p_{IL}) dxdydz} \qquad \text{Eq. 5-3}$$

where

$$L(N, r_1, r_2, r_3 \mid p_I, p_L, p_{IL}) = \frac{N!}{r_1! r_2! r_3! (N - r_1 - r_2 - r_3)!} (p_I - p_{IL})^{r_1} (p_L - p_{IL})^{r_2} p_{IL}^{r_3} (1 + p_{IL} - p_I - p_L)^{N - r_1 - r_2 - r_3} \qquad \text{Eq. 5-4}$$

is the multinomial likelihood of the observation (r_1, r_2, r_3, N).

A similar model has been used in the past in assessing reliability of various systems built with components (Littlewood, Popov et al. 2000), (Popov 2002).

For the comparison of the DBMSs we define the following criterion:

Probability of an *inadequate response*, P_{Ser}, by the DBMS, which represents the event of getting either an incorrect or a late response. Clearly, $P_{Ser} = P_I + P_L - P_{IL}$. Its posterior distribution, $f_{p_{Ser}}(\bullet \mid N, r_1, r_2, r_3)$, can be derived from the joint posterior, $f_{p_I, p_L, p_{IL}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$, by first transforming it to, for example, $f_{p_I, p_L, p_{Ser}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$, and then integrating out the nuisance parameters P_I and P_L.

An often used selection method (Port and Chen 2004) in the literature is the weighted sum of the values of the attributes. The weighted sum of the two attributes in our study can be calculated as follows: $P_S = kP_I + (1-k)P_L$, in which the constant k is defined by the assessor. High values of k correspond to cases when incorrect results are highly undesirable while late results may be tolerable. On the contrary, low values of k correspond to cases when incorrect results may be tolerated by the system while late responses may have serious consequences. In order to derive the marginal distribution of P_S first, the joint distribution $f_{p_I, p_L, p_{IL}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$ is transformed to $f_{p_I, p_L, p_S}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$ and then the nuisance parameters P_I and P_L are integrated out, as we did above for P_{Ser}. However, we will not be using this method of selection since the new variable P_S does not have an obvious intuitive meaning. The difficulty is compounded in our case since the uncertainty is stated explicitly. It is impossible to say what a confidence of say 99% associated with a particular value of P_S tells us about the component being assessed.

5.4. A Numerical Example

In this section we demonstrate the approach of selecting a DBMS based on the evaluation of the respective performance and reliability attributes. In total six database servers from four different vendors were used. Four of the servers are open-source, namely PostgreSQL 7.0, PostgreSQL 7.2, Interbase 6.0 and Firebird 1.0. The other two servers are commercial closed development servers, anonymised here due to the restrictive ‘End User License Agreements’. We refer to these components as CS1 (Commercial Server 1) and CS2 as they are from different vendors.

An ideal selection of a database server based on the results of statistical testing may be problematic in practice. We highlight two circumstances in which these difficulties can occur:

- Assume that we are interested in choosing between several DBMSs, based on their performance. The ideal situation for choosing the most appropriate database server based on measurements *after* deployment is clearly *unrealistic* since we would like to select the best server *before* the application is developed.
- Assume that DivRep integrator would like to make a *strategic* choice of a database server for use in the foreseeable future. In this scenario the application(s), which may be developed in the future may be even unknown at the time of making the selection; therefore performing statistical testing (which is crucially dependent on knowing the operational profile in the targeted environment) will be impossible.

Given these difficulties we can use an alternative option, i.e. we can use well-known benchmark applications. In the context of database servers this might be TPC-C benchmark. In this case, the performance of the components can be measured directly on the target platform, but there might be problems observing failures. This is because it would be reasonable to expect that a database server would correctly process the operations defined in the benchmark application (despite different implementations of the operations being possible). Thus, in this case the selection of the database server would be significantly influenced by the performance attribute. Even if failures are observed, such a measurement of the reliability may be very expensive; the likely candidates to choose from will be reliable components. In that case the amount of testing to observe a few failures may be prohibitively high (Adams 1984).

We illustrate the assessment method with data collected from experiments with our own implementation of TPC-C benchmark (Section 2.4). In the empirical study we recorded response times of each transaction. The test harness was similar to the one used for the experiments described in previous sections. It consisted of two machines:

- A server machine, on which one of the six database servers was run.
- A client machine, which executed a Java implementation of TPC-C standard.

Each experiment comprised the *same sequence* of 1000 transactions. We ran two types of experiments:

- *Single client* - a TPC-C compliant client modifies the database by executing the specified transaction mix.

- *Multiple clients* - a TPC-C compliant client modifies the database and additional 10 clients concurrently execute read-only transactions (Order-Status and Stock-Level). The choice of the workload was made in order to, straightforwardly, avoid database inconsistencies caused by inherent non-determinism of the different servers and prevent conflicts of concurrent transactions.

Multiple clients experiment enabled us to increase the load on the servers and measure the effect of the increased load on their performance. A timeout value, specific to each transaction type, was used to distinguish between late and timely responses. We defined two sets of timeouts:

- The 90th percentile values specified by TPC-C (*TPC-C timeout*).
- One fifth of the 90th percentile values (*short timeout*).

The choice of the timeout values was made after a personal communication with a TPC-C affiliate and auditor who confirmed that the values were conservative for a wide range of on-line transaction processing applications (see Section 2.4). We defined four scenarios, varying the number of clients and timeout values respectively:

- Scenario 1 - single client / TPC-C timeouts.
- Scenario 2 - single client / short timeouts.
- Scenario 3 - multiple clients / TPC-C timeouts.
- Scenario 4 - multiple clients / short timeouts.

The database servers were compared for each of the scenarios.

5.4.1. Prior Distributions

The prior, $f_{p_I, p_L, p_{IL}}(\bullet,\bullet,\bullet)$, was constructed under the assumption that P_I and P_L are independently distributed random variables, i.e. $f_{p_I, p_L}(\bullet,\bullet) = f_{p_I}(\bullet) f_{p_L}(\bullet)$. We made this assumption since we did not have any objective evidence to believe otherwise. In case there are reasons (objective or subjective) then the assumption of independence may be dropped. In this case the particular form of $f_{p_I, p_L}(\bullet,\bullet)$ should be defined explicitly. Additionally, the conditional distributions $f_{p_{IL}|P_I, P_L}(\bullet \mid P_I, P_L)$ were defined for every pair of values of P_I and P_L, in the range [0, min(P_I, P_L)] since the probability of incorrect and late responses cannot be greater than the probability of *either* of the two individually. In passing we note that the choice of the prior is not critical here since with this benchmark application an arbitrarily large number of

demands can be generated, which can correct any inaccuracies of the priors, i.e. 'the data will speak for itself'.

We anticipated observing mainly late responses while the incorrect result failures were expected to be very rare. We have assumed 'ignorance prior' (Uniform distribution) for performance in the range in the range $P_L \in [0,1]$. For incorrect result failures we have also assumed ignorance but using an upper bound of 10^{-2}, likely to be very conservative in the context of TPC-C, i.e. we used the range $P_I \in [0,10^2]$. We assumed ignorance priors for both P_I and P_L since we did not have any preference regarding their values. In this study we used the same distribution for all the servers since for the scenarios tested we did not have any reason to prefer one server over the others. There might, however, be cases – some discussed later in Section 5.5 - whereby the assessor may have different prior beliefs about the competing servers.

A summary of the distributions used and the range in which they are defined is given in Table 5-2.

Table 5-2 The Prior distributions (identical for all six servers and all four scenarios).

Prior Distribution	Range	Distribution Type
Reliability $f_{p_I}(\bullet)$	0 – 0.01	Uniform
Performance $f_{p_L}(\bullet)$	0 – 1	Uniform
Conditional distribution: $f_{p_{IL} \mid p_I, p_L}(\bullet \mid P_I, P_L)$	0 – min(P_I, P_L)	Uniform

5.4.2. Observations

The observations from the TPC-C experiments are given in Table 5-3. The number of demands for all servers is 1000. Five out of six servers exhibit late result failures only. Incorrect result failures are observed only for CS2. In addition, whenever a result was incorrect on CS2 it was late, too. The incorrect results observed were due to the specific concurrency control mechanism used by CS2 (Popov, Strigini et al. 2004). The locks on resources, e.g. database rows, were not released properly when the lock holding transactions were completed. To resolve the problem we had to install timeout watchdogs and abort transactions when the timeout expired. Each aborted transaction was repeated as many times as necessary to eventually commit successfully. We decided to use *transaction repetition count* as the criterion of an

incorrect response on CS2. In particular, we defined a threshold of 5 as the value, beyond which the transaction would be considered to have failed.

We used transaction timeout values and *transaction repetition count* to classify each demand on each server in the categories r_1 to r_4 (defined in Section 5.3).

Table 5-3 The observations of the six database servers for the four scenarios. The number of demands (N) is 1000 for each server. We did not observe any incorrect-only failures, i.e. r_1=0 for all servers.

DBMS	Scenario 1			Scenario 2			Scenario 3			Scenario 4		
	r_1	r_2	r_3	r_1	r_2	r_3	r_1	r_2	r_3	r_1	r_2	r_3
PG 7.0	0	1	0	0	30	0	0	0	0	0	644	0
PG 7.2	0	6	0	0	33	0	0	3	0	0	489	0
IB 6.0	0	0	0	0	24	0	0	1	0	0	434	0
FB 1.0	0	0	0	0	1	0	0	0	0	0	439	0
CS1	0	0	0	0	33	0	0	19	0	0	303	0
CS2	0	0	0	0	4	0	0	0	1	0	329	1

5.4.3. Posteriors

The percentiles derived from the posterior distribution for the 4 scenarios are given in Table 5-4. One can see that the ordering between the servers changes as the number of clients and/or the timeout values vary (to improve the readability of the table we have explicitly shown the ranking order of the servers in each scenario).

Under Scenario 1 (the least demanding scenario) four servers (IB 6.0, FB 1.0, CS1 and CS2) produce identical results since they completed without any failure (i.e. on time and correctly) the 1000 transactions. We are indifferent in the choice among them. The two versions of PostgreSQL exhibit late responses and they are ranked lowest.

When we decrease the timeout value (Scenario 2) the ranking changes: now there are late responses with all the servers. The two worst servers are PostgreSQL 7.2 and CS1. Interestingly, Firebird 1.0, an open-source server, is ranked the best.

In Scenario 3 the percentile values are close again as in the first scenario, though the earlier version of PostgreSQL, PG 7.0, is ranked the best, alongside Firebird 1.0 while CS1 is the worst performing server. Firebird 1.0 is consistently among the best servers in the first 3 scenarios. An interesting observation is the 50th percentile value of the posteriors CS2 and IB 6.0. Even though the total number of failures for these two servers were the same (1 each, see Table 5-3), the nature of the failure was different:

the result from CS2 was both incorrect and late whereas from IB 6.0 it was only late. Exploring this dependence we can still see a difference in the 50th percentile values of these two servers (even though the difference is marginal and on the chosen accuracy of expressing the percentile values is not observed in the 99th percentile).

The ranking changes again in the most demanding scenario (Scenario 4). The best server is now CS1.

Table 5-4 Percentiles (abbreviated to P-tile) for the distribution of the system quality $P_{Ser} = P_I + P_L - P_{IL}$ classified per scenario. To improve the readability we have also provided the Ranking order for each of the servers based on the percentile values. The prior distribution is the same for all servers across all scenarios.

P-tile	COTS	Prior	Scenario 1		Scenario 2		Scenario 3		Scenario 4	
			Posterior	Rank	Posterior	Rank	Posterior	Rank	Posterior	Rank
0.5	PG 7.0	0.502	0.0021	5	0.0310	4	0.0012	1	0.6436	6
	PG 7.2		0.0071	6	0.0340	5	0.0041	5	0.4888	5
	IB 6.0		0.0012	1	0.0250	3	0.0021	4	0.4340	3
	FB 1.0		0.0012	1	0.0021	1	0.0012	1	0.4392	4
	CS1		0.0012	1	0.0340	5	0.0200	6	0.3032	1
	CS2		0.0012	1	0.0051	2	0.0020	3	0.3300	2
0.99	PG 7.0	0.992	0.0076	5	0.0456	4	0.0060	1	0.6780	6
	PG 7.2		0.0152	6	0.0492	5	0.0108	5	0.5256	5
	IB 6.0		0.0060	1	0.0384	3	0.0076	3	0.4704	3
	FB 1.0		0.0060	1	0.0076	1	0.0060	1	0.4756	4
	CS1		0.0060	1	0.0492	5	0.0324	6	0.3376	1
	CS2		0.0060	1	0.0124	2	0.0076	3	0.3652	2

The results of the ranking apply to the chosen servers, and may change if different versions are used, e.g. the ordering between PostgreSQL 8.1.4 and Firebird 2.0.1 is different, as shown in Section 4.4.

We can see that under the more ‘stressful’ profile in Scenario 4 the best DBMS is CS1. However if the concurrency is low, then even with the rigid performance requirements (Scenario 2) Firebird 1.0 server, which is open-source and freely available, comes out as the best server. On the other hand the worst components are PostgreSQL servers. We could also use the outcome of the study as a validation of the proposed method. CS1, which came out best, is widely accepted as the best DBMS and has by far the largest share in the market of database servers. This gives some confidence that the data that we used is sufficiently informative to allow for meaningful and accurate discrimination between the competing components and the

method itself is trustworthy to provide rigorous ground for accurate component selection.

5.5. Discussion and Related Work

To assess the attributes of interest of DivRep components in a single framework and handle the inherent uncertainty in the assessment we propose the use of “uncertainty explicit” methods. We have illustrated in the previous subsections how the assessment of the two attributes, performance and reliability, can be performed for ranking of DBMS products. The ranking can be used for selecting components of DivRep middleware - two servers that turn out to be the “best” according to the ranking would be selected. This is not ideal because it is possible that two servers, which scored best individually do not form the best pair, e.g. the servers’ performance might be different once the replication protocol is in place. It is worth noting that the described method is not intrinsically related to DivRep and it can be applied in a wider context where need for similar ranking of other products arises.

In the example of the assessment method (Section 5.4) we have used somewhat arbitrary definitions of incorrect response failures. A better alternative for more representative assessment of reliability attribute can be found in (Gashi, Popov et al. under review), where a set of faults (bugs) from four different database severs form a demand space, which is used for more *stressful* testing of the components under assessment. More interestingly, the results obtained from the related study also led to CS1 being the best server. This gives an extra confidence that the method indeed performs plausibly, i.e. does not lead to counterintuitive results, which would have required further scrutiny to explain why the perception of CS1 as the best is not supported by the results of the study.

Using the same approach Gashi et al. (Gashi and Popov 2007) show how an optimal pair of components can be selected to form 1-out-of-2 fault-tolerant system. The focus of the paper is on the assessment of the reliability and not performance. They use the same Bayesian model described here, applied to a dataset of faults reported for four database servers, to choose the pair with the lowest probability of the coincident failure. In this way the authors cater for the possibility that the best pair might not be built with the database servers which are the best individually (indeed they confirmed this possibility). However the approach is applied to running the servers on their own,

i.e. a reported fault is executed against an individual server. It would be problematic to apply the approach to performance only – an overhead due to the replication (various synchronisations) may lead to performance significantly different from when the servers are run on their own. In any case, the approach can be combined with what is presented in this section with the aim to select the "best pair" taking into account both fault logs and performance.

The definition of the prior distribution is fundamental in Bayesian assessment. In our study we have assumed that prior distributions for each component are the same. This was due to the unavailability of other known 'objective' evidence that we could use to define more accurate priors. Anecdotal evidence about the servers does exist, but is difficult to translate these subjective beliefs into priors in the form required by our method. By assuming that the prior distributions were the same for each server, the decision on which server is chosen is dictated by the observations only. As a result the decision of the types of distributions for the random variables in our study becomes less important.

However there are other ways of deriving more accurate priors. We could, for example, utilize evidence from *earlier versions* of the servers and then do multiple steps of inference, i.e. if we want to perform the assessment with later versions of the servers in our study (e.g. with versions of PostgreSQL after release 7.2 or Firebird after release 1.0) we can use the posteriors derived here as priors for the later versions, collect the new evidence for the later versions and then use the model to derive the posteriors for each. This approach has also been reported elsewhere (Littlewood and Wright 1997).

The method of assessment proposed in this paper would be applicable to different families of COTS components. The setup described in Section 5.3 and illustrated in Section 5.4 is particularly relevant for COTS components with *stringent reliability and performance* requirements. We provided empirical results using off-the-shelf database servers. Java Virtual Machines (JVMs), various application servers, web servers and Business process execution engines (Andrews, Curbera et al. 2003) are also examples of COTS components were reliability and performance requirements are usually the deciding attributes for selection. Fault and failure reports, which can be used as observations, do exist for these products and so do performance benchmarks (e.g. *ab* benchmarking tool for web servers (ApacheSoftwareFoundation 2008)). Therefore, similar measurements to what we did for database servers are also possible

with these other families of COTS components. In many cases for these components one may not need to deal with more than 2 attributes, which makes our 2-attribute model proposed in Section 4 immediately applicable without any further simplifications.

6. Related Work

Every extension of knowledge arises from making the conscious the unconscious.

Friedrich Nietzsche

Earlier sections discussed different topics related to the research and each of them contains the references to the relevant work. In this section, we describe in detail the work related to a particular topic, database replication, in order to emphasize it as one of the central themes in the research work.

6.1. A Multitude of Database Replication Solutions

Database replication is a thoroughly studied subject. The main challenge of database replication is replica-control and coordination of modification operations (DELETE, INSERT and UPDATE). As mentioned in Section 2.3, the well-known paper (Gray, Helland et al. 1996) categorised database replication solutions according to the *place* where replication is performed (*primary copy* and *update all* approaches) and the *point in time* when the replicas coordinate the updates to the database (*eager* and *lazy* replications). Eager solutions (Kemme and Alonso 2000a), (Pedone and Frolund 2000), (Kemme, Bartoli et al. 2000) guarantee that the writes are propagated to all replicas before transaction *commit*. This has a negative impact on system performance, mainly because the scalability of the solution is limited, but ensures database consistency in a straightforward way. Lazy solutions (Daudjee and Salem 2004), (Liskov, Rivka et al. 1992), (Pedone, Guerraoui et al. 1997), (Pacitti, Minet et al. 2001), (Amza, Cox et al. 2003) perform writes after commit. They offer improved performance at the possible expense of compromising database consistency. Typically, reconciliation techniques have to be deployed once the states of replicated databases diverge. Moreover, hybrid solutions, which combine the characteristics of both eager and lazy solutions, have been proposed in the literature. For example, (Elnikety, Dropsho et al. 2006) describes *Tashkent*, a database replication solution, which on one hand uses eager certification technique to identify global *write-write*

conflicts in the system to provide replica consistency, while on the other hand once a transaction has been certified, its writeset is propagated lazily to the remote replicas, i.e. the remote changes are executed in separate transactions.

Numerous database replication solutions have been proposed in academia and industry. In order to minimise the problems described in (Gray, Helland et al. 1996) the solutions have been divided in regard to the degree of separation of the replica control from the underlying database systems. The first approach is so called *black-box* approach where replication mechanism is provided completely outside of the database server. These solutions are usually middleware-based (Plattner and Alonso 2004), (Patino-Martinez, Jimenez-Peris et al. 2005), (Cecchet, Marguerite et al. 2004) where further concurrency control is performed in an additional software layer. On contrary, there exist replication solutions that extend the internals of database servers in order to provide replica control. These solutions are referred to as *kernel-based* or similarly, *white box* solutions (Kemme and Alonso 2000a), (Kemme and Wu 2005), (Pedone, Guerraoui et al. 2003). There are examples of database replication which use features of both approaches. The work of (Patino-Martinez, Jimenez-Peris et al. 2005) is an example of a mixture of *black-box* and *white-box* approach, where specific features of the database servers are combined with an external mechanism for concurrency control. Middleware-based solutions are easier to develop than white box approaches, which depend on knowing the internals of database servers. The former can be maintained independently from the database servers they operate on. However abort rates pose a problem in the middleware-based techniques since only partial knowledge of the transactional conflicts is available. A lack of the real implementation is a disadvantage of many replication solutions despite, usually, a sound theoretical foundation (Kemme and Alonso 2000b), (Elnikety, Zwaenepoel et al. 2005). Nonetheless there exist examples of advanced implementations for database replication such as the one developed under C-JDBC (Cecchet, Marguerite et al. 2004) project. C-JDBC is an open source database cluster middleware designed for high performance and improved fault tolerance. It features a load balancer that works under one of the three types of algorithms: round-robin, weighted round-robin or least pending request first. A query results cache is maintained and there are several optimisations of transaction performance: 1) *lazy transaction begin,* 2) *parallel transaction execution* and 3) *early response to update commit,* or *abort*. The last two optimisations are similar to the processing of DivRep. C-JDBC executes write,

commit and abort (Section 2.4.2 in (Cecchet, Marguerite et al. 2004)) operations sequentially, while in DivRep a more relaxed replica-control algorithm executes only commits and begins serially. In terms of fault-tolerance C-JDBC provides checkpointing, recovery logs and *horizontal scalability*, which prevents the system becoming a single point of failure. However it does not have any potential for protection against design faults as is the case in DivRep. Sequoia project (Continuent 2007), a descendant of C-JDBC, offers clustering, load balancing and failover services for heterogeneous databases.

Further classification of eager database replication protocols can be found in (Weismann, Pedone et al. 2000). Using three key parameters (server architecture, server interaction and transaction termination) the authors identify eight classes and for each of them examine respective requirements, capabilities and cost. The *pessimistic* regime of DivRep performs a form of active replication (Wiesmann, Pedone et al. 2000). Early variants of active replication can be found in (Garcia-Molina and Pitelli 1989) and (Keidar 1994). The performance of an active replication and its comparison to two-phase commit solutions is given in (Amir and Tutu 2002).

Several strands of research have been developing eager replication solutions using group communication (Powell 1996). The early work of (Hadzilacos and Toueg 1993), (Agrawal, Alonso et al. 1997) and (Schiper and Raynal 1996) has provided the foundation for proliferation of such solutions. The fundamental semantics of group communication exploited in the database replication are ordering and atomicity of message delivery and group maintenance. In particular, total order of message delivery guaranteed by the group communication systems is used to ensure consistent execution of transactions on all replicas. In (Agrawal, Alonso et al. 1997) a few variants of replica management protocols using group broadcasts are presented. Firstly a protocol that uses the state machine approach (Schneider 1982) where operations are executed in the same order on all replicas is explained. Using atomic broadcast, a transaction broadcasts all operations to the remaining sites and they install all the conflicting operations in the same order – the one imposed by the broadcast. To improve the performance of the protocol reads are localized and thus the overhead imposed by sending them to all remote sites is alleviated. As a final enhancement the authors decrease the number of messages exchanged per transaction to one – all writes are bundled into one message and sent in the end of transaction to the remote sites. Once the locks are granted for the writes and they are executed, the

transaction commits. Order of conflicting operations is guaranteed to be the same by the properties of the underlying communication system and thus there is no need for broadcasting commits.

A comprehensive comparison of database replication techniques based on total order broadcast can be found in (Wiesmann and Schiper 2005). The techniques had been compared separately in other studies, but usually only against a classic replication scheme like distributing locking. The comparison is done in the same environment using the same settings. A sound comparison of different protocols is given in (Alvisi, Elnozahy et al. 2002), though the authors are concerned with rollback and recovery and not database replication.

The commercial solutions usually give up consistency for better performance (Oracle 2005). There are, however, commercial solutions for eager replication. Oracle DataGuard (Oracle 2002) replication uses primary copy approach where production database is accompanied with two types of standby databases, *physical standby* database and *logical standby* database. The former has identical on-disk database structures as the primary database while the latter is an independent database that contains the same data as the primary. *Logical standby* database is updated using SQL operations and is frequently used for recovery purposes. The consistency between the production database and the physical standby is maintained using online redo log services, *Log Transport Services* and *Log Apply Services*. Similar replication solution is offered by the DB2 HADR system. Ingres Replicator (Ingres 2006) is an example of multi-master solution. It offers both full and partial replication and uses a 2-Phase commit protocol to ensure atomicity of distributed transactions. There have been several projects that are developing database replication solutions using the open source database server PostgreSQL, such as PgPool, Sequoia, Slony and PGCluster. They provide either update all or primary copy replication and some of them use variants of 2-phase commit protocol. Postgres-R project, originally developed by Prof Bettina Kemme (Kemme 2000) as a part of the PhD research, is another example of a replication solution based on PostgreSQL.

All mentioned replication solutions deal with fault-tolerance in replicated databases on the crash failure assumption, i.e. none of them are able to tolerate subtle faults that, for example, cause databases to diverge or report incorrect results to client. One exception to this is *commit barrier scheduling (CBS)* (Vandiver, Balakrishnan et al. 2007), a recently proposed replication protocol that guards against Byzantine faults in

transaction processing systems. The work is built on the research of (Castro and Liskov 2002), (Castro, Rodrigues et al. 2003) and the early scheme for tolerating Byzantine faults in database systems (Garcia-Molina, Pittelli et al. 1986). The authors propose an implementation of CBS that ensures single-copy serializable view to clients to be used in Heterogonous Replicated Database (HRDB). In contrast to DivRep no possibility to handle multi-version concurrency control mechanisms that guarantee snapshot isolation is possible in HRDB. As in DivRep the use of diverse (the authors of (Vandiver, Balakrishnan et al. 2007) use term *heterogeneous* instead) databases increases the possibility of detecting, and recovering, from non-fail-stop failures. Their results about the potential for dependability improvement through diverse redundancy are consistent with the ones reported in (Gashi, Popov et al. 2007) – out of 251 tested bugs, most of them (131) caused incorrect answers, database corruption or unauthorised access. HRDB is a primary copy replication scheme and thus the performance of the replicated system is dictated by the processing of the primary replica. To address the non-fail-stop failures HRDB postpones the commit of a transaction until after majority of replicas have agreed on the results. Additionally HRDB deals with the non-determinism of the locking mechanisms in diverse replicas by requiring that a primary is *sufficiently blocking* – if the primary executes a pair of SQL operations in parallel, it is guaranteed that all non-faulty secondaries will be able to do so, too. In the cases where a primary is not *sufficiently blocking* performance penalty might be observed, e.g. in a system consisting of 3 replicas if the primary is not sufficiently blocking for the faster secondary, the commit will be executed only once the slower secondary is ready and not after the faster secondary has reached the end of the transaction. In DivRep conflicts are detected by both replicas, although different transactions might be identified as “victims”. There is no need for a primary: liveness is guaranteed by the use of *NOWAIT* parameter, without the need for *transaction-ordering* rule and *sufficiently blocking* property to hold and, moreover, there is a possibility to improve performance of the replicated system by exploiting systematic differences between the diverse replicas.

6.2. Load Balancing and Adaptability

The research of (Zuikeviciute and Pedone 2006) proposes conflict-aware load-balancing technique for certification-based replication protocols. The technique is

aimed at satisfying two requirements: increasing concurrency and simultaneously decreasing the abort rate of certification-based protocols placed at the middleware level. Commonly, one has to use opposing techniques to meet these requirements. The authors propose a hybrid load-balancing technique which unifies two approaches *Maximizing Parallelism First (MPF)*, aimed at promoting high degree of parallelism for transaction execution and distributing the load as evenly as possible between replicas, and *Minimizing Conflicts First (MCF)*, which tries to assign conflicting transaction to the same replica so that the replica's scheduler serializes the offending operations, instead of waiting for the *certification* test to abort the transactions. An example with TPC-C implementation is given and the results show that for a particular workload scenario abort rate can be reduced without a high penalty in response time.

Milan-Franco et al. (Milan-Franco, Jimenez-Peris et al. 2004) address the issue of adaptability of a middleware-based replication to the changes imposed by different transactional profiles and loads submitted to the replicas. For that matter dynamically adaptive *multiprogramming levels* (MPL), which set the number of concurrently executing transactions in a database system, are proposed. The authors suggest a two-part solution to deal with the adaptability, *local-level adaptation*, which focuses on maximizing the concurrency degree on each individual replica, and *global-level adaptation*, which strives to improve the overall performance of the system by distributing the load fairly between all replicas. The work recognizes two principal goals for the dynamically adaptable middleware; the first one is identification of the optimal MPL so that the admission of additional transactions is impossible once the database system reaches the limit in number of simultaneous executions, and the second one is a provision for an adjustment in MPL once a change in the load is observed. This work differs from the previous research in that achieving optimal throughput is sought using the techniques implemented in the middleware.

On the other hand, in (Brown, Mehta et al. 1994) a feedback-based algorithm, *M&M*, is proposed, which is used to find optimum MPL and memory settings for a particular type of workload class, so as to achieve the response time goals. The algorithm relies on information from the internals of the system to build the algorithm, such as disk buffer controller. *M&M* algorithm and the adaptive distributed middleware (Milan-Franco, Jimenez-Peris et al. 2004) are complementary to the functions of DivRep and both could be used to improve system performance.

Elnikety et al. (Elnikety, Dropsho et al. 2006) propose a middleware-based replication solution that joins two properties of transaction execution, durability and transaction ordering, so as to achieve better scalability and performance. In particular, the need for the unification of the two properties in a replicated system is followed from the practice observed in centralised databases, where multiple commits are grouped into a single action and written to the disk in a single operation for increased efficiency. Related configuration parameters, devised for performance improvement of standard transactional logging performed by WAL (Write-Ahead Logging) component, exist in PostgreSQL; they are *commit_siblings* and *commit_delay*. The authors propose two types of solutions to unify the properties in a replicated system, where ordering is guaranteed by the middleware while durability is provided by the individual replicas. The first one, *Tashkent-MW*, is a pure middleware solution which disables synchronous writes on the replicas and thus enhances the middleware with a feature to log database writes. The second one, *Tashkent-API*, is an analogous solution that extends the API of the underlying replicas with a feature that specifies the commit order of the update transactions. Their results show that for different types of workloads, ranging from update-only to read-oriented, both *Tashkent-MW* and *Tashkent-API* exhibit better performance than a base system, in which transaction ordering and durability are achieved separately.

Tashkent+ (Elnikety, Dropsho et al. 2007) is an enhanced version of *Tashkent* that features a memory-aware load balancing algorithm (MALB). The algorithm uses estimates of memory consumption for each type of transaction through querying the respective execution plans to obtain information about tables and indexes used. These working set estimates are used to group transactions so that their combined usage of the memory can be optimally satisfied e.g. by allocating them to the least loaded replicas. This effectively partitions groups of transactions across replicas. MALB uses the knowledge to introduce an *update filtering* technique and reduce the load of update transactions. This technique allows some tables to be dropped or their content to become stale on some replicas, so the updates destined for the particular tables are not processed. *Tashkent+* is developed on assumption that transactions and their respective parameters used in an application are known in advance. Authors report superior performance of MALB when compared to other load balancing techniques. They demonstrate it using varying workloads, memory and database sizes.

6.3. Consistency Guarantees

The effects of snapshot isolation (SI) on a centralized database have been studied in (Fekete, Liarokapis et al. 2005) and (Fekete, O'Neil et al. 2004). The authors propose a sufficient condition that guarantees serializable histories under SI. They show how an application can be modified to satisfy the condition. Fekete et al. (Fekete 2005) discuss the possibilities of allocating different isolation levels to a group of transactions while still maintaining conflict serializability. Each subgroup of transactions uses a distinct concurrency control mechanism to ensure the isolation level of choice. The authors provide a simple graph-based algorithm for determining the weakest acceptable allocation when transactions on both serializable and snapshoot isolation level are executing jointly.

The work of (Elnikety, Zwaenepoel et al. 2005) defines *generalised snapshot isolation* (GSI) for replicated databases (Section 2.3.2). A special case of GSI, *prefix-consistent snapshot isolation* (PCSI), is presented as a suitable isolation level for replicated databases. Instead of ensuring that each transaction operates on the latest snapshot available on any of the replicas, PCSI guarantees that only the most recent changes performed on the local database are available. Two possible implementations are proposed: centralized and distributed certification. The former employs a master database, which communicates with all replicas to certify the commits of the modifying transactions. In this way only the master is guaranteed to have the latest snapshot of the data. To alleviate the issue with the master database being a single point of failure, the authors proposed distributed certification in which each replica executes and certifies update transactions. This functionality is supported by the use of an atomic broadcast for delivery of writesets, necessary information in certification procedure, to all replicas. The work of (Beeri, Bernstein et al. 1989) has proposed *order-preserving serializability* to address the differences between correctness criteria defined in classical serializability theory and temporal precedence. The property has been originally proposed for proving the correctness of nested transactions. An execution is *order-preserving serializable* if it is serializable and the following holds: whenever T_i commits before T_j begins, i.e. the operations of T_i execute before operations of T_j, then there exists an equivalent serial execution in which T_i precedes T_j. Concurrency control algorithm using 2-phase locking produces order-preserving serializable executions. (Alonso 1997) discusses the use of the *order-preserving*

serializability as the criterion for guaranteeing consistency in partially replicated database, and in particular for the systems where group communication primitives are used. It gives an example that ensuring serializable schedules on all replicas in a partially replicated system is not sufficient to guarantee global serializability. This is due to the fact that serializability does not necessarily follow temporal precedence. As a result 1-copy serializability is not order-preserving. Schmidt et al. (Schmidt and Pedone 2007) analyse a common technique, known as *deferred updates*, for propagating writes in replicated databases. The authors define a conceptual deferred update protocol and formally characterize it. They prove that a weaker property than *order-preserving serializable* is needed to ensure that replication protocols using deferred update technique guarantee database consistency. They denote the new property as *active order-preserving serializable*.

7. Conclusions

This is not the end. It is not even the beginning of the end.
But it is, perhaps, the end of the beginning
Winston Churchill

Diverse redundancy was originally conceived as an incarnation of fault-tolerance 30 years ago. Due to the high cost, however, the industry has been reluctant to use diverse redundancy for guaranteeing sufficient levels of dependability. It was only recently, as a consequence of proliferation of numerous off-the-shelf software, that the use of diversity has become a realistic possibility for dependability improvement. Nevertheless the results of assessing the potential dependability gains are still scarce. The situation is even more acute when we consider performance – to the best of the author's knowledge no research that tries to use design diversity for performance improvement has been reported elsewhere. Consequently, the research described in this thesis is concerned with the implications of diverse redundancy (when multiple heterogeneous products are deployed in a redundant configuration) on the performance. We have chosen database servers, a concrete and mature type of computer system, as the basis for the research. The research work has lead to several major contributions:

- From the practical point of view a middleware-based database replication solution has been devised and an algorithm that ensures database consistency has been developed and proved correct.
- Also, a previously proposed Bayesian model (Littlewood, Popov et al. 2000) has been adapted for selection of an optimal database server for inclusion in the FT-node. We have taken two relevant attributes, performance and reliability, into account when demonstrating the selection.
- From the empirical perspective we provided a thorough evaluation of the use of diverse database servers for database replication.

Consequently, the whole research described in this thesis can be regarded as an initial step toward building a fully operational fault-tolerant SQL server.

7.1. Research Assessment

We have implemented a configurable database replication middleware (DivRep), deployable with diverse database servers, to allow the clients to request quality of service in line with their specific requirements for performance and dependability. Clients with conflicting needs between dependability assurance and performance may benefit from diverse redundancy according to their own priorities; because an FT-node can apply different regimes for different clients. When performance is the top priority the *optimistic* regime, which tends to execute read operations only on the faster server, can be used. In many practical cases this is likely to produce significant improvement. On the other hand, when dependability is the top priority, the *pessimistic* regime with a fully featured middleware for fault-tolerance provides significant level of dependability assurance, unattainable if traditional non-diverse replication is used. To overcome the opposing requirements dictated by dependability and performance we proposed a hybrid solution that combines two regimes of operation to offer a *configurable quality of service*. This hybrid solution is one of the reasons why we measured the performances of the two extremes, *pessimistic* and *optimistic* regimes of operation. By deploying *run-time assessment capabilities*, based on Bayesian inference, we propose that the middleware may process the individual SQL operations switching intelligently between different regimes of operation.

A novel replication algorithm, Dependable Replication Algorithm (DRA), guarantees strong consistency between the replicas preserving *conventional snapshot isolation* (Elnikety, Zwaenepoel et al. 2005) as found in centralised databases. It, also, ensures *strict 1-copy-SI*, a consistency criterion based on *1-copy-SI* (Lin, Kemme et al. 2005). This form of consistency is not present in ROWA replication, where read operations might not observe the latest modifications in the replicated system. DRA was conceived with *simplicity* in mind - little complexity is added to concurrency managers of DBMSs to preserve consistency among replicated databases, e.g. no pre-declaration of objects used in a transaction or knowledge of write-sets is needed. Thus, no artificial dependencies between transactions are possible.

Under the loads with high degree of concurrency, imposing total order of transaction boundary operations to ensure database consistency might cause a performance bottleneck - clients block each other due to contention for execution of transaction boundaries. To overcome this performance overhead we propose a contention-aware

scheduling at the middleware level that manipulates the priorities of the database processes. By favouring the database processes that execute transaction boundaries we show that prioritisation policy brings performance improvements when CPU of database servers is *underutilised*, despite the extra work imposed due to adjusting priorities of database processes.

We have presented some encouraging empirical results which suggest that diversity can improve the performance of the fault-tolerant solution. This possibility is due to the fact that different database server may "complement" each other, as we have established empirically for two open-source servers: one of the servers is systematically faster in processing some types of transactions while the other server is faster processing other types of transactions. This is similar to the intuitive idea of forming teams of individuals who have different skills, which is seen as a strength of the teams in many areas.

We compared diverse with non-diverse redundancy using an optimistic architecture of FT-node and observed that the differences between the execution times of the diverse replicas are turned into a performance advantage. In those experiments, executed under a specific workload, diverse redundancy is clearly beneficial compared with non-diverse redundancy. In addition we also compared non-replicated solutions (a single copy of a DBMS) with an FT-node, in which a diverse pair of DBMSs is deployed. Diverse pair performed significantly faster than the non-replicated solutions (Stankovic and Popov 2006).

Using a pair of diverse servers, the performance obtained with the two regimes of operation of DivRep had been compared to a simulated version of a ROWA scheme. Similarly to ROWA, the simulation uses static load-balancing optimised at the expense of tolerating crash failures only. The results show that the performance penalty due to the use of diverse redundancy via DivRep can be significant. Despite being obtained for specific workloads, the results are instructive for those who are keen on deploying diverse redundancy, but have concerns about performance.

It seems quite clear that increasing the number of replicas will make the issue more severe (Gray, Helland et al. 1996) – further performance deterioration would ensue. Therefore, while scaling the replication protocol is possible (DRA is not limited to the use of two replicas), its usefulness is doubtful. We envisage that DivRep is used to build fault-tolerant node (FT-node) with only two database servers, capable of tolerating non-crash failures. FT-nodes can then be combined with a high

performance replication scheme (eager or, even, a lazy one). Such a hybrid replication scheme will allow one to have both – high dependability assurance (via the FT-nodes) and improved performance (via the properties of the scalable replication scheme).

The results showed that the difference between the performances of DivRep and the simulated version of a ROWA scheme can be attributed to the difference in the individual performances of the diverse servers deployed in DivRep: one of the DBMS products turns out to be significantly slower than the other on the selected suite of application profiles (note, however, that the performance ranking between the DBMSs is variable – a multitude of the experiments revealed that the ranking changes depending on the particular choice of products' versions). This led us to the important conclusion that the selection of database servers for deployment in DivRep should take into account both, the ability to deliver dependability assurance as well as the ability to minimise performance penalty.

To address this issue we provide formalism for the selection procedure, in which both dependability assurance and performance are taken into account. Using an adaptation of the Bayesian model defined in (Littlewood, Popov et al. 2000) we assess servers according to the two attributes, reliability and performance, in order to choose the best one.

We have also shown that, under the provisions of *user-centric analysis*, the transaction response times are shorter with DivRep than with the simulated replication protocol based on ROWA approach. The results have been observed under different types of workloads. The analysis demonstrates how the differences in performance of diverse servers can be used for decreasing the transaction latencies in DivRep.

7.2. Future Directions

Besides the contributions presented in the preceding section, the work described in the thesis gives rise to several possibilities for future work:

- Implementation of the *hybrid* regime of operation of DivRep. An assessment method for dynamically switching between the dependability- and performance-oriented regime of operation needs be developed.
- Extending the evaluation of DivRep using other isolation levels. Although relevant, snapshot isolation is not the default isolation level in most DBMSs. For example, further experimentation could be performed, using read committed

isolation level, which is the mostly used one in practice (MicrosoftSQL, Oracle and PostgreSQL specify it as the default isolation level).

- Potentially improving the performance of diverse database replication, and especially the *pessimistic* i.e. dependability-oriented regime of operation, by employing other replication techniques, e.g. *deferred writes*.

  One of the issues that would have to be revisited, if DivRep is to employ deferred writes technique, is the way to guarantee the consistency of the databases after modifying operations are executed. Using standard log-shipping or write-set extraction as a part of deferred writes replication, the results of modifying operations, executed on a replica, are propagated and applied on others without consistency checks. In this way error detection becomes problematic - there is no possibility to adjudicate the results produced by different replicas. New approaches to overcome this limitation are needed in order to make the error detection viable. Only then the replication technique could be used so it potentially improves the performance of the *pessimistic* regime. A straightforward alternative to making error detection possible is to propagate SQL operations instead of the changes made by the writes. In this case, however, the performance benefits would be less likely.
- Devising a "fusion" replication protocol that integrates FT-node in a high-performing replication solution and provides improved dependability (through FT-node capabilities for error detection) as well as high scalability (through the use of scalability properties of the underlying replication protocol).
- Developing the necessary middleware components for users to be able to try out data replication with diverse DBMSs despite the issues with differences in SQL dialects (e.g. in this way the replication with DivRep will be transparent to the client applications). Deficiency of these components is a practical obstacle in the way of the adoption and practical evaluation of these solutions. Some DBMS vendors have recently made a move in that direction: EnterpriseDB Advanced Server (EnterpriseDB 2006) and Fyracle (Janus-Software 2006) are Oracle-mode implementations based on PostgreSQL and Firebird DBMS engines, respectively. Also, there are a couple of software products developed for migration between different databases: SQLFairy (SQLFairy 2007) and SwissSQL (SwisSQL 2007). Both tools offer the possibility to port schema (DDL operations) and data between different DBMSs. Additionally, SwisSQL provides for conversion of database

"run-time" aspects – SQL queries and stored procedures can be translated between different dialects. With these solutions the problem with SQL dialects is reduced.

- Further experimental studies for assessing the potential of diverse redundancy for both dependability and performance improvement of COTS software are needed. In the research work we have scrutinised only a particular replication protocol when evaluating the performance implications of using diverse redundancy for database replication. Furthermore, the evaluation has been performed with only two open-source database servers. Thus, there is no definitive judgement of implications of design diversity on system performance. Firstly, the experimental evaluation can be extended by using other DBMSs, of different complexity, with or without the particular replication protocol, to measure the potential of design diversity further. A consequence of the extended evaluation would be beneficial for assessment of the replication protocol, too. More generally, there is a myriad of other COTS software that can be scrutinised, e.g. web-servers, application servers etc., for evaluation of usefulness of diverse redundancy in regard to dependability and performance.

Bibliography

R. K. Abbott and H. Garcia-Molina (1992). "Scheduling Real-Time Transactions: A Performance Evaluation." ACM Transactions on Database Systems (TODS) 17(3): 513-560.

E. N. Adams (1984). "Optimizing Preventive Service of Software Products." IBM Journal of Research and Development 28(1): 2-14.

A. Adya, B. Liskov and P. O'Neil (2000). Generalized Snapshot Isolation Levels. IEEE International Conference on Data Engineering (ICDE), San Diego, CA.

D. Agrawal, G. Alonso, A. El Abbadi and I. Stanoi (1997). Exploiting Atomic Broadcast in Replicated Databases. In Proceedings of EuroPar (EuroPar'97), Passau (Germany).

A. Ailamaki, D. T. McWherter, B. Schroeder and M. Harchol-Balter (2004). Priority Mechanisms for OLTP and Transactional Web Applications. 20th International Conference on Data Engineering, Boston, USA.

G. Alonso (1997). Partial Database Replication and Group Communication Primitives. 2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97), Valais, Switzerland.

L. Alvisi, E. N. Elnozahy, Y. Wang and D. Johnson (2002). "A Survey of Rollback-Recovery Protocols in Message-Passing Systems." ACM Computing Surveys (CSUR) 34(3): 374-408.

Y. Amir and C. Tutu (2002). From Total Order to Database Replication. 22nd International Conference on Distributed Computing Systems (ICDCS'02), IEEE Computer Society.

P. E. Ammann and J. C. Knight (1988). "Data Diversity: An Approach to Software Fault Tolerance." IEEE Transactions on Computers 37(4): 418-425.

C. Amza, A. Cox and W. Zwaenepoel (2003). Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. ACM/IFIP/Usenix International Middleware Conference, Rio de Janeiro, Brazil.

T. Anderson and P. A. Lee (1990). Fault Tolerance: Principles and Practice. New York, Springer-Verlag.

T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte and I. Trickovic (2003). Business Process

Execution Language for Web Services version 1.1. 2003, BEA, IBM, Microsoft, SAP, Siebel, Systems http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

ANSI (1992). ANSI X3.135-1992, American National Standard for Information Systems.

ApacheSoftwareFoundation (2008). ab - Apache HTTP server benchmarking tool, http://httpd.apache.org/docs/2.0/programs/ab.html. 2008.

A. Avizienis and L. Chen (1978). N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. IEEE Eighth Annual International Symposium on Fault-Tolerant Computing Systems, FTCS-8, Toulouse, France.

A. Avizienis and J. P. J. Kelly (1984). "Fault Tolerance by Design Diversity: Concepts and Experiments." IEEE Computer 17(8): 67-80.

C. Babbage (1974). On the Mathematical Powers of the Calculating Engine (Unpublished Manuscript, December 1837). The Origins of Digital Computers: Selected Papers. B Randell, Springer: 17-52.

C. Beeri, P. A. Bernstein and N. Goodman (1989). "A Model for Concurrency in Nested Transactions Systems." Journal of the ACM (JACM) 36(2): 230-269.

A. Behm, S. Rielau and R Swagerman (2004). Returning Modified Rows - SELECT Statements with Side Effects. Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, Morgan Kaufmann.

H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil and P. O'Neil (1995). A Critique of ANSI SQL Isolation Levels. SIGMOD International Conference on Management of Data, San Jose, California, United States, ACM Press New York, NY, USA.

P. Bernstein and N. Goodman (1981). "Concurrency Control in Distributed Database Systems." ACM Computing Surveys 13(2): 185-221.

P. Bernstein, V. Hadzilacos and N. Goodman (1987). Concurrency Control and Recovery in Database Systems. Reading, Massachusetts, Addison-Wesley.

Borland (1999). Interbase 6.0 Operations Guide, Borland - The Open ALM Company: 314.

Borland (2007). Borland - The Open ALM Company.

P.B. Bovet and M. Cesati (2005). Understanding the Linux Kernel, O'Reilly.

S. Brocklehurst, Y. Chan, B. Littlewood and J. Snell (1990). "Recalibrating Software Reliability Models." IEEE Transactions on Software Engineering 16(4): 458-470.

P. K. Brown, M. Mehta, J. M. Carey and M. Livny (1994). Towards Automated Performance Tuning for Complex Workloads. Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

N. Budhiraja, K. Marzullo, F. B. Schneider and S. Toueg (1993). The Primary-Backup Approach. Distributed Systems. S. Mullender, ACM Press: 199-216.

M. J. Carey, R. Jauhari and M. Livny (1989). Priority in DBMS Resource Scheduling. Proceedings of the 15th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc.

M. Castro and B. Liskov (2002). "Practical Byzantine Fault Tolerance and Proactive Recovery." ACM Transactions on Computer Systems (TOCS) 20(4): 398-461.

M. Castro, R. Rodrigues and B. Liskov (2003). "BASE: Using Abstraction to Improve Fault Tolerance." ACM Transactions on Computer Systems (TOCS) 21(3): 236-269

E. Cecchet, J. Marguerite and W. Zwaenepoel (2004). C-JDBC: Flexible Database Clustering Middleware. USENIX Annual Technical Conference, Freenix.

P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz and D. A. Patterson (1994). "RAID: High-Performance, Reliable Secondary Storage." ACM Computing Surveys (CSUR) 26(2): 145-185.

Continuent (2007). Sequoia Project - Database Clustering Technology. 2007.

C.J. Date (1994). An Introduction to Database Systems, Addison-Wesley.

K. Daudjee and K. Salem (2004). Lazy Database Replication with Ordering Guarantees. 20th International Conference on Data Engineering.

K. Daudjee and K. Salem (2006). Lazy Database Replication with Snapshot Isolation. Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, VLDB Endowment

J. M. Dickey (1982). Conjugate Families of Distributions. Encyclopedia of Statistical Sciences. S. Kotz and N. L. Johnson. New York, Wiley. 2: 135-145.

D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk and J. P. J. Kelly (1991). "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability." IEEE Transactions on Software Engineering 17(7): 692-702.

S. Elnikety, S. Dropsho and F. Pedone (2006). Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, Leuven, Belgium, ACM New York, NY, USA.

S. Elnikety, S. Dropsho and W. Zwaenepoel (2007). Tashkent+: Memory-Aware Load Balancing and Update Filtering in Replicated Databases. EuroSys 2007, Lisbon, Portugal.

S. Elnikety, W. Zwaenepoel and F. Pedone (2005). Database Replication Using Generalised Snapshot Isolation. Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), IEEE Computer Society.

EnterpriseDB (2006). http://www.enterprisedb.com/.

A. Fekete (2005). Allocating Isolation Levels to Transactions. Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Baltimore, Maryland, ACM New York, NY, USA.

A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil and D. Shasha (2005). "Making Snapshot Isolation Serializable." ACM Transactions on Database Systems (TODS) 30(2): 492-528.

A. Fekete, E. O'Neil and P. O'Neil (2004). "A Read-Only Transaction Anomaly Under Snapshot Isolation." ACM SIGMOD Record 33(3): 12-14.

H. Garcia-Molina and F. Pitelli (1989). "Reliable Scheduling in a TMR Database System." ACM Transactions on Computer Systems (TOCS) 7(1): 25-60.

H. Garcia-Molina, F. Pittelli and S. Davidson (1986). "Applications of Byzantine Agreement in Database Systems." ACM Transactions on Database Systems (TODS) 11(1): 27-47.

I. Gashi (2007). Software Dependability With Off-The-Shelf Components. Centre for Software Reliability. London, City University: 278.

I. Gashi, P. Bishop, B. Littlewood and D. Wright (2007). Reliability Modelling of a 1-Out-Of-2 System: Research with Diverse Off-The-Shelf SQL Database Servers. 18th International Symposium on Software Reliability Engineering (ISSRE '07), Trollhattan, Sweden, IEEE Computer Society Press.

I. Gashi and P. Popov (2006). Rephrasing Rules for Off-The-Shelf SQL Database Servers. Sixth European Dependable Computing Conference, 2006. EDCC '06.

I. Gashi and P. Popov (2007). Uncertainty Explicit Assessment of Off-The-Shelf Software: Selection of an Optimal Diverse Pair. Proceedings of the Sixth International Conference on COTS Based Software Systems, ICCBSS-2007, Banff, Alberta, Canada, IEEE Computer Society Press.

I. Gashi, P. Popov and V. Stankovic (under review). "Uncertainty Explicit Assessment of Off-The-Shelf Software." Elsevier Information and Software Technology Journal.

I. Gashi, P. Popov, V. Stankovic and L. Strigini (2004). On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers. Architecting Dependable Systems II. R. de Lemos, C. Gacek and A. Romanovsky, Springer. LNCS 3069**:** 191-214.

I. Gashi, P. Popov and L. Strigini (2007). "Fault Tolerance via Diversity for Off-The-Shelf Products: A Study With SQL Database Servers." IEEE Transactions on Dependable and Secure Computing 4(4): 280-294.

A. Gorbenko, V. Kharchenko, P. Popov and A. Romanovsky (2005). Dependable Composite Web Services with Components Upgraded Online. Architecting Dependable Systems III. R. de Lemos, C. Gacek and A. Romanovsky, Springer. LNCS 3549**:** 92-121.

J. Gray (1978). Notes on Database Operating Systems. Operating Systems: An Advanced Course, Springer Verlag, Berlin. Lecture Notes in Computer Science**:** 393-482.

J. Gray (1981). The Transaction Concept: Virtues and Limitations. 7th International Conference on Very Large Data Bases (VLDB), Cannes, France, IEEE Computer Society.

J. Gray, P. Helland, D. Shasha and P. O'Neil (1996). The Dangers of Replication and a Solution. ACM SIGMOD International Conference on Management of Data, Montreal, Canada, SIGMOD.

J. Gray and L. Lamport (2006). "Consensus on Transaction Commit." ACM Transactions on Database Systems (TODS) 31(1): 133-160.

J. Gray, R. Lorie, G. Putzolu and I. Traiger (1975). Granularity of Locks and Degrees of Consistency in a Shared Data Base. IFIP Working Conference on Modelling of Data Base Management Systems Freudenstadt.

J. Gray and A. Reuter (1993). Transaction Processing: Concepts and Techniques, Morgan Kaufmann.

R. Guerraoui (1995). Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. Proceedings of the International Workshop on Distributed Algorithms (WDAG '95), Le Mont Saint Michel.

V. Hadzilacos and S. Toueg (1993). Fault-Tolerant Broadcast and Related Problems. Distributed Systems. S. Mullander, Addison-Wesley**:** 97-145.

C. Hall and P. Bonnet (2005). Getting Priorities Straight: Improving Linux Support for Database I/O. Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway.

Hewlett-Packard (2005). HP Integrity Superdrome - TPC-C Executive Summary, Hewlett-Packard**:** 3.

M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte and G. T. Wong (2000). Survivability through Customization and Adaptability: The Cactus Approach. DARPA Information Survivability Conference & Exposition.

IBM (2007). DB2 Product Overview - SQL Data-Change Operations. 2007.

Ingres (2006). Replicator Option User Guide: 233.

ISO (2003). ISO/IEC 9075-1: 2003 Information Technology - Database Languages - SQL - Part 1: Framework (SQL/Framework).

R. Jain (1991). The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. New York, NY, Wiley-Interscience.

Janus-Software (2006). Fyracle, http://www.janus-software.com/fb_fyracle.html.

J. Jeanrenaud and P. Romanazzi (1994). Software Product Evaluation: A Methodological Approach. Proceedings of Software Quality Management II: Building Software into Quality.

R. Jimenez-Peris, M. Patino-Martinez, G. Alonso and B. Kemme (2003). "Are Quorums an Alternative for Data Replication?" ACM Transactions on Database Systems 28(3): 257-294.

I. Keidar (1994). A Highly Available Paradigm for Consistent Object Replication. Institute of Computer Science. Jerusalem, The Hebrew University of Jerusalem: 52.

J. P. Kelly and A. Avizienis (1983). A Specification-Oriented Multi-Version Software Experiment. 13th Fault-Tolerant Computer Symposium (FTCS), Milan, Italy.

B. Kemme (2000). Database Replication fo Clusters of Workstations. Swiss Federal Institute of Technology. Zurich: 145.

B. Kemme and G. Alonso (2000a). Don't Be Lazy, Be Consistent: Postgres-R, a New Way to Implement Database Replication. International Conference on Very Large Databases (VLDB), Cairo, Egypt.

B. Kemme and G. Alonso (2000b). "A New Approach to Developing and Implementing Eager Database Replication Protocols." ACM Transactions on Database Systems (TODS) 25(3): 333-379.

B. Kemme, A. Bartoli and O. Babaoglu (2000). Online Reconfiguration in Replicated Databases Based on Group Communication. Bologna, University of Bologna: 15.

B. Kemme and S. Wu (2005). Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. International Conference on Data Engineering, Tokyo, Japan, IEEE Computer Society.

J. C. Knight and N. G Leveson (1986). "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming." IEEE Transactions on Software Engineering 12(1): 96-109.

J. Kontio, S. Y. Chen, K. Limperos, R. Tesoriero, G. Caldiera and M. Deutsch (1995). A COTS Selection Method and Experiences of Its Use. Twentieth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, Maryland.

W. B. Lampson and D. D. Redell (1980). "Experience with Processes and Monitors in Mesa." Communications of the ACM 23(2): 105-107.

J. C. Laprie, C. Béounes and K. Kanoun (1990). "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures." IEEE Computer 23(7): 39-51.

J. C. Laprie, B. Randell , A. Avizienis and C. Landwehr (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing." IEEE Transactions on Dependable and Secure Computing 1(1): 11-33.

Y. Lin, B. Kemme, M. Patino-Martinez and R. Jimenez-Peris (2005). Middleware Based Data Replication Providing Snapshot Isolation. ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, ACM Press.

B. Liskov, L. Rivka, S. Liuba and G. Sanjay (1992). "Providing High Availability Using Lazy Replication." ACM Transactions on Computer Systems (TOCS) 10(4): 360-391.

B. Littlewood, P. Popov and L. Strigini (2000). Assessment of the Reliability of Fault-Tolerant Software: A Bayesian Approach. International Conference on Computer Safety, Reliability and Security (SAFECOMP '00), Rotterdam, The Netherlands, Springer.

B. Littlewood, P. Popov and L. Strigini (2001). "Modelling Software Design Diversity - A Review." ACM Computing Surveys 33(2): 177-208.

B. Littlewood and D. Wright (1997). "Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software." IEEE Transactions on Software Engineering 23(11): 673-683.

M. R. Lyu (1995). Software Fault Tolerance. New York, NY, USA, John Wiley & Sons, Inc.

M. R. Lyu (1996). Handbook of Software Reliability Engineering, McGraw-Hill and IEEE Computer Society Press.

Microsoft (2000). Transact-SQL.

Microsoft (2004). Implement a Continuously Updating, High-Resolution Time Provider for Windows.

J. Milan-Franco, R. Jimenez-Peris, M. Patino-Martinez and B. Kemme (2004). Adaptive Middleware for Data Replication. Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware, Toronto, Canada, Springer-Verlag, New York.

J. D. Musa (1993). "Operational Profiles in Software-Reliability Engineering." IEEE Software 10(2): 14-32.

C. Ncube and N. Maiden (1999). PORE: Procurement Oriented Requirements Engineering Method for the Component-Based Systems Engineering Development Paradigm. Proceedings of the International Workshop on Component-Based Software Engineering.

Oracle (2002). Oracle Data Guard: Ensuring Disaster Recovery for the Enterprise**:** 20.

Oracle (2005). Oracle 10g Advanced Replication: Replication Manual
246.

E. Pacitti, P. Minet and E. Simon (2001). "Replica Consistency in Lazy Master Replicated Databases." Distributed and Parallel Databases 9(3).

C. Papadimitriou (1986). The Theory of Database Concurrency Control. New York, Computer Science Press.

M. Patino-Martinez, R. Jimenez-Peris, B. Kemme and G. Alonso (2005). "MIDDLE-R: Consistent Database Replication at the Middleware Level." ACM Transactions on Computer Systems (TOCS) 23(4): 375-423.

F. Pedone and S. Frolund (2000). Pronto: A Fast Failover Protocol for Off-The-Shelf Commercial Databases. 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), Nuremberg, Germany, IEEE.

F. Pedone and S. Frolund (2005). Pronto: High Availability for Standard Off-The-Shelf Databases. Lausanne, EPFL.

F. Pedone and R. Guerraoui (1997). On Transaction Liveness in Replicated Databases. Proceedings of IEEE Pacific Rime International Symposium on Fault-Tolerant Systems (PRFTS'97), IEEE Computer Society, Washington, DC, USA.

F. Pedone, R. Guerraoui and A. Schiper (1997). Transaction Reordering in Replicated Databases. 16th IEEE Symposium on Reliable Distributed Systems (SRDS'97), Durham, NC, IEEE.

F. Pedone, R. Guerraoui and A. Schiper (2003). "The Database State Machine Approach." Distributed and Parallel Databases 14(1): 71-98.

C. Plattner and G. Alonso (2004). Ganymed: Scalable Replication for Transactional Web Applications. 5th ACM/IFIP/USENIX International Middleware Conference, Toronto, Canada

P. Popov (2002). Reliability Assessment of Legacy Safety-Critical Systems Upgraded with Off-The-Shelf Components. SAFECOMP'2002, Catania, Italy, Springer-Verlag.

P. Popov, L. Strigini, A. Kostov, V. Mollov and D. Selensky (2004). Software Fault-Tolerance with Off-The-Shelf SQL Servers. 3rd International Conference on COTS-Based Software Systems, ICCBSS'04, Redondo Beach, CA USA, Springer.

P. Popov, L. Strigini and A. Romanovsky (2000). Diversity for Off-The-Shelf Components. International Conference on Dependable Systems & Networks (FTCS-30, DCCA-8) - Fast Abstracts, New York, NY, USA.

D. Port and Z. Chen (2004). Assessing COTS Assessment: How Much is Enough? Inetrnational Conference on COTS Based Software Systems, ICCBSS '04, Redondo Beach, California, Springer-Verlag.

PostgreSQL (2007). PostgreSQL 8.1.10 Documentation, Chapter 33. Triggers.

D. Powell (1996). "Group Communication." Communications of the ACM 39(4): 50-53.

F. Raab (2005). TPC-C 90th Percentile Response Time Constraints. .

B. Randell (1975). "System Structure for Software Fault Tolerance." IEEE Transactions on Software Engineering 1(2): 220-232.

M. R. Rinard and P. C. Diniz (2003). "Eliminating Synchronization Bottlenecks Using Adaptive Replication." ACM Transactions on Programming Languages and Systems 25(3): 316-359.

A. Schiper and M. Raynal (1996). "From Group Communication to Transactions in Distributed Systems." Communications of the ACM 39(4): 84-87.

R. Schmidt and F. Pedone (2007). A Formal Analysis of the Deferred Update Technique. 11th International Conference on Principles of Distributed Systems (OPODIS 2007), Guadeloupe, French West Indies.

F. B. Schneider (1982). "Synchronization in Distributed Programs." ACM Transactions on Programming Languages and Systems 4(2): 125-148.

D. Skeen (1981). Nonblocking Commit Protocols. Proceedings of ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, ACM Press, New York, NY, USA.

D. Solomon and M. Russinovich (2000). Inside Windows 2000, Microsoft Press, U.S.

SQLFairy (2007). SQLFairy - SQL Translator. 2007.

J. Stankovic, H. S. Son and J. Hansson (1999). "Misconceptions About Real-Time Databases." IEEE Computer 32(6): 29-36.

V. Stankovic and P. Popov (2006). Improving DBMS Performance through Diverse Redundancy. 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06), Leeds, United Kingdom, IEEE Computer Society.

SwisSQL (2007). SwisSQL - Data Migration. 2007.

M. Thakur (1994). Transaction Models in InterBase 4. Proceedings of the Borland International Conference.

TPC (2002a). TPC Benchmark C, Standard Specification, Version 5.0., Transaction Processing Performance Council.

TPC (2002b). TPC Benchmark W (Web Commerce), Standard Specification, Version 1.8, Transaction Processing Performance Council.

TPC (2007). TPC Benchmark H (Decision Support), Standard Specification, Revision 2.6.1, Transaction Processing Performance Council.

A. Valdes, A. Almgren, S. Cheung, Y. Deswarte, B. Duterte, J. Levy, H. Saidi, V. Stavridou and T. E. Uribe (2003). An Architecture for an Adaptive Intrusion-Tolerant Server. Lecture Notes in Computer Science (LNCS) 2845 - Selected Papers from 10th Int. Workshop on Security Protocols 2002, Springer.

B. Vandiver, H. Balakrishnan, B. Liskov and S. Madden (2007). Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles, Stevenson, Washington, USA, ACM New York, NY, USA

J. von Neumann (1956). Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. Automata Studies. C. E. Shannon and J. McCarthy, Princeton University Press: 43-98.

M. Weismann, F. Pedone and A. Schiper (2000). Database Replication Techniques: A Three Parameter Classification. 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), Nurnberg, Germany, IEEE.

M. Wiesmann, F. Pedone, A. Schiper, B. Kemme and G. Alonso (2000). Understanding Replication in Databases and Distributed Systems. Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000), Taipei, Taiwan, IEEE Computer Society Los Alamitos.

M. Wiesmann and A. Schiper (2005). "Comparison of Database Replication Techniques Based on Total Order Broadcast." IEEE Transactions on Knowledge and Data Engineering 17(4): 551-566.

Wikipedia (2007). Multi-Master Replication, Wikipedia, The Free Encyclopedia.

A. Wool (1998). "Quorum Systems in Replicated Databases: Science or Fiction?" Data Engineering Bulletin 21(4): 3-11.

D. Wright and K. Y. Cai (1994). Representing Uncertainty for Safety Critical Systems. London, Centre for Software Reliability, City University: 135.

Y. Zhuge, H. Garcia-Molina and J. L. Wiener (1998). "Consistency Algorithms for Multi-Source Warehouse View Maintenance." Distributed and Parallel Databases 6(1): 7-40.

V. Zuikeviciute and F. Pedone (2006). Conflict-Aware Load-Balancing Techniques for Database Replication. Lugano, University of Lugano.

List of Acronyms

2PC – Two-Phase Commit Protocol

2PL – Two-Phase Locking

BPEL - Business Process Execution Language

COTS – Commercial-Off-The-Shelf

CSI – Conventional Snapshot Isolation

DBMS – Database Management System

DML – Data Manipulation Language

DRA – Dependable Replication Algorithm

FB – FireBird Database Management System

GSI – Generalised Snapshot Isolation

JDBC – Java Database Connectivity

MSSQL – Microsoft SQL Database Management System

MVCC – Multi-Version Concurrency Control

NB-AC - Non-Blocking Atomic Commitment

OLAP – On-Line Analytical Processing

OLTP – On-Line Transaction Processing

PCSI – Prefix-Consistent Snapshot Isolation

PG – PostgreSQL Database Management System

RAID - Redundant Arrays of Independent (or Inexpensive) Drives

ROWA – Read-Once-Write-All

ROWAA – Read-Once-Write-All-Available

S2PL – Strict Two-Phase Locking

SG – Serialization Graph

SI – Snapshot Isolation

TPC – Transaction Processing Performance Council

TPC-C – TPC’s Benchmark C

Appendix A

Database Schema of the Log Database

```
CREATE TABLE [dbo].[InconsistentResults] (
	[id] [int] IDENTITY (1, 1) NOT NULL ,
	[results] [text] NULL ,
	[SQL] [varchar] (1024) NULL ,
	[clientID] [int] NULL ,
	[txnID] [int] NULL
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

CREATE TABLE [dbo].[LogCCs] (
	[LCC_ID] [int] IDENTITY (1, 1) NOT NULL ,
	[Data] [varchar] (2040) NULL ,
	[StartDate] [datetime] NULL ,
	[FinishDate] [datetime] NULL ,
	[StartTime] [numeric](18, 0) NULL ,
	[FinishTime] [numeric](18, 0) NULL ,
	[ConType] [int] NULL
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[LogClientSQLs] (
	[LCQ_CLIENT_ID] [int] NOT NULL ,
	[LCQ_LCT_ID] [int] NOT NULL ,
	[LCQ_ID] [int] NOT NULL ,
	[Data] [varchar] (2040) NULL ,
	[StartTime] [bigint] NULL ,
	[FinishTime] [bigint] NULL
```

```
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[LogDataLoader] (
	[ID] [int] IDENTITY (1, 1) NOT NULL ,
	[CLoad] [int] NULL ,
	[CRun] [int] NULL ,
	[CTime] [datetime] NULL ,
	[Seed] [int] NULL ,
	[ConType] [int] NULL ,
	[Warehouses] [int] NULL
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[LogExperimentParameters] (
	[LEP_ID] [int] IDENTITY (1, 1) NOT NULL ,
	[NUMBER_OF_CLIENTS] [int] NOT NULL ,
	[EXPERIMENT_TYPE] [int] NULL ,
	[REPETITION_ID] [int] NULL ,
	[NUMBER_OF_TRANSACTIONS] [int] NULL ,
	[SEED] [int] NULL ,
	[NUMBER_OF_BUFFERS] [int] NULL ,
	[SERVER_1] [int] NULL ,
	[SERVER_2] [int] NULL ,
	[SKIP] [bit] NULL ,
	[NUMBER_OF_WHOUSES] [int] NULL ,
	[PURGE_THRESHOLD] [int] NULL
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[LogServerSQLs] (
	[LQ_CLIENT_ID] [int] NOT NULL ,
	[LQ_LCT_ID] [int] NOT NULL ,
	[lq_id] [int] NOT NULL ,
```

```
	[CONTYPE] [int] NOT NULL ,
	[StartTime] [bigint] NULL ,
	[FinishTime] [bigint] NULL ,
	[StartResultFetchTime] [bigint] NULL ,
	[FinishResultFetchTime] [bigint] NULL ,
	[Type] [tinyint] NULL
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[LogTransactions] (
	[LCT_CLIENT_ID] [int] NOT NULL ,
	[LCT_ID] [int] NOT NULL ,
	[Data] [varchar] (2040) COLLATE NULL ,
	[StartDate] [datetime] NULL ,
	[FinishDate] [datetime] NULL ,
	[StartTime] [bigint] NULL ,
	[FinishTime] [bigint] NULL ,
	[conType] [int] NULL ,
	[exception] [tinyint] NULL ,
	[isPause] [bit] NULL ,
	[purgePeriod] [tinyint] NULL ,
	[T_ID] [int] NULL
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[SQLType] (
	[Type] [int] NOT NULL ,
	[description] [varchar] (64) NULL
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[exception] (
	[E_ID] [tinyint] NOT NULL ,
	[server_type] [char] (8) NULL ,
```

```
    [sql_error_code] [int] NULL ,
    [sql_state] [varchar] (128) NULL ,
    [Description] [varchar] (4096) NULL
) ON [PRIMARY]
GO
```