View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne

GROUP COMMUNICATIONS AND DATABASE REPLICATION: TECHNIQUES, ISSUES AND PERFORMANCE

THÈSE N^{O} 2577 (2002)

PRÉSENTÉE À LA FACULTÉ I&C, SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Matthias WIESMANN

informaticien diplômé de l'Université de Genève
de nationalité suisse et originaire de Collonge-Bellerive (GE)

acceptée sur proposition du jury:

Prof. A. Schiper, directeur de thèse
Prof. K. Aberer, rapporteur
Prof. G. Alonso, rapporteur
Prof. Ph. Pucheral, rapporteur

Lausanne, EPFL
2002

Abstract

Databases are an important part of today's IT infrastructure: both companies and state institutions rely on database systems to store most of their important data. As we are more and more dependent on database systems, securing this key facility is now a priority. Because of this, research on fault-tolerant database systems is of increasing importance.

One way to ensure the fault-tolerance of a system is by replicating it. Replication is a natural way to deal with failures: if one copy is not available, we use another one. However implementing consistent replication is not easy. Database replication is hardly a new area of research: the first papers on the subject are more than twenty years old. Yet how to build an efficient, consistent replicated database is still an open research question.

Recently, a new approach to solve this problem has been proposed. The idea is to rely on some communication infrastructure called *group communications.* This infrastructure offers some high-level primitives that can help in the design and the implementation of a replicated database. While promising, this approach to database replication is still in its infancy.

This thesis focuses on group communication-based database replication and strives to give an overall understanding of this topic. This thesis has three major contributions. In the structural domain, it introduces a classification of replication techniques. In the qualitative domain, an analysis of fault-tolerance semantics is proposed. Finally, in the quantitative domain, a performance evaluation of group communication-based database replication is presented.

The classification gives an overview of the different means to implement database replication. Techniques described in the literature are sorted using this classification. The classification highlights structural similarities of techniques originating from different communities (database community and distributed system community). For each category of the classification, we also analyse the requirements imposed on the database component and group communication primitives that are needed to enforce consistency.

Group communication-based database replication implies building a system from two different components: a database system and a group communication system. Fault-tolerance is an end-to-end property: a system built from two components tends to be as fault-tolerant as the weakest component. The analysis of fault-tolerance semantics show what fault-tolerance guarantee is ensured by

group communication based replication techniques. Additionally a new fault-tolerance guarantee, *group-safety*, is proposed. Group-safety is better suited to group communication-based database replication. We also show that group-safe replication techniques can offer improved performance.

Finally, the performance evaluation offers a quantitative view of group communication based replication techniques. The performance of group communication techniques and classical database replication techniques is compared. The way those different techniques react to different loads is explored. Some optimisation of group communication techniques are also described and their performance benefits evaluated.

Résumé

Les bases de données représentent une composante importante de l'infrastructure informatique d'aujourd'hui. Que ce soit dans le monde de l'industrie ou de l'administration, de plus en plus d'institutions utilisent des bases de données pour stocker les informations cruciales. Chaque jour, notre société dépend de manière croissante des bases de données. La fiabilisation de ces systèmes est donc devenu une priorité. De ce fait, les bases de données tolérantes aux fautes sont un domaine de recherche dont l'importance va croissant.

Une manière d'offrir la tolérance aux pannes est la réplication. La réplication est une approche naturelle aux problèmes des défaillances: si une copie des données ne fonctionne plus, on a recours à une autre copie. Malheureusement, assurer une réplication consistante du point de vue des données n'est pas chose aisée. La recherche sur les bases de données répliquées n'est de loin pas nouvelle: les premières publications sur le sujet fêteront bientôt leur quart de siècle. Toutefois, la question d'une technique de réplication de base de données consistante et efficace reste aujourd'hui encore ouverte.

Récemment, une nouvelle approche au problème a été proposée : elle s'appuie sur une infrastructure de communication généralement appelée *communications de groupes*. Cette infrastructure permet, au moyen de primitives de haut niveau, de simplifier l'architecture et l'implémentation de techniques de réplication. Bien que prometteuse, cette approche n'en est qu'à ses premier balbutiements et la plupart des recherches dans ce domaine n'en sont qu'à un stade exploratoire.

Cette thèse se concentre sur les techniques de réplication de bases de données basées sur les communications de groupes. Le but est d'obtenir une certaine compréhension du domaine et de fournir une synthèse. Cette thèse comporte trois contributions majeures. Du point de vue structurel, elle introduit une classification des techniques de réplication. Du point de vue qualitatif, une analyse des garanties de tolérance aux pannes est proposée. Finalement, une analyse de performance offre un point de vue quantitatif sur le sujet.

La classification donne un aperçu des différentes approches possibles au problème de la réplication de bases de données. Les différentes techniques proposées dans la littérature sont classifiées et cataloguées. Cette classification met en évidence les similarités structurelles entre des techniques originant de communautés scientifiques différentes (communuauté des basses de donnée et communauté de systèmes répartis). Pour chaque catégorie de techniques, les impératifs imposés

au sous-système de base de donnée ainsi que les primitives de communications nécessaires pour assurer la cohérence sont présentés.

Les techniques de bases de données basées sur les communications de groupes s'appuient sur deux composants : un module de communication de groupes et un module de base de données. La tolérance aux pannes est une propriété globale : un système construit à partir de deux composants aura tendance à être aussi robuste que le plus faible de ces deux composants. L'analyse des garanties de tolérance aux fautes permet de comprendre quel niveau de robustesse est offert par une technique de réplication basée sur des communications de groupes. Un nouveau critère de tolérance aux pannes est proposé: « *group-safety* ». Ce critère est plus approprié pour décrire la tolérance aux pannes des techniques de réplication basées sur des communications de groupes. De plus, certaines techniques peuvent être adaptées à ce critère, et ainsi offrir des performances améliorées.

Finalement, l'évaluation de performances donne une vue quantitative des performances de techniques de réplication basées sur des communications de groupes. Les performances de ces techniques, ainsi que de techniques de réplication classiques, sont comparées. Le comportement de toutes les techniques à des charges différentes est étudié et présenté. Enfin des optimisations liées à certaines techniques basées sur les communications de groupe sont présentées, et leurs bénéfices en termes de performance sont évalués.

In Memoriam Isabelle de Bellet

Acknowledgements

I am very grateful to many people who helped in one way or another for this work. First of all, I would like to thank André Schiper for his confidence in me, for accepting me in his research group and supervising me in this work.

I also wish to thank all people involved in the DRAGON project: Fernando Pedone, Bettina Kemme, Gustavo Alonso and Roel Vandewall, all people without whom this work would not have been possible.

I would also like to express my gratitude to all the people in the Operating System Lab and the Distributed Systems Lab. In particular, I would like to thank Péter Urbán and Xavier Défago, for always being available to discuss issues from distributed system algorithm to programming bugs, Philippe Altherr for the interesting discussions on software architecture and programming languages, Arnas Kupšys, Paweł Wojciechowski and Sergio Mena de la Cruz for their support.

I would also like to thank both secretaries, Kristine Verhamme and France Faille for the logistical support they gave me and Sylvie Roux for proof-reading the final version.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Tout a déjà été dit, mais comme personne n'écoutait, il faut recommencer.
>
> **André Gide**

The turn of the century has seen tremendous advances in the field of computing. This has brought two major changes. First, the number of computer systems has increased by an order of magnitude. From desktop systems to embedded circuitry, computers have become ubiquitous. Secondly, the emergence of networking infrastructure, both local and global, means that computers are increasingly interconnected.

Changes did not only happen in the technological field: our societies depend more and more on computers: each day, an activity starts to rely on a computer. Already, aspects of everyday's life that do not imply computer systems at one stage or another are very rare. Today, the impact of computers on society is more important than ever.

Yet, as computer system become more widespread, complex and distributed, they also become more fragile. Failures are becoming both more frequent and more serious in terms of consequences. Crashes, bugs and other problems have an increasingly important impact on human lives [Neu02]. Issues like fault-tolerance and security were considered specialised fields for a long time. Nowadays, our society relies more and more on computers, and people become aware of this fact. So research on security and fault-tolerance becomes more and more important. Even mainstream software vendors have started to make security and stability their official priority [Man02].

Many approaches to build fault-tolerant systems have been proposed. They reach all aspects of information systems: from hardened material components to improved management of IT projects. Techniques that promise fault-tolerance include software engineering, specialised computer languages and replication. In this thesis we will concentrate on replication.

Replication is a natural way to deal with failures: if one replica fails, another takes over. Most living organisms achieve fault-tolerance this way: if one cell dies, the other cells share the work so that the organism as a whole can survive. Yet, while replication seems a natural solution for fault-tolerance, implementing a consistent replicated system is not easy.

The main issue is that computer system store data in a way that is very different from biological entities. Another issue is that the expectations are different. If a neuron dies in the brain, no memory is lost, but the information becomes more "fuzzy". This is not an acceptable solution for many computer systems. In safety critical system that control factories or power-plants, the slightest error can have dire consequences. Even for banking applications, having "slightly wrong" accounts because of a server crash is not an acceptable solution. If we want our system to enforce strict guarantees even in case of crash, replication is a complex problem.

Banking systems are a typical example of database applications. In general, databases represent a key element of the IT infrastructure of many companies: payrolls, customer information and inventories are stored in databases. Because of this, replicating databases to increase their fault-tolerance appears to be a good idea. Yet, consistent data replication is a complex issue [Cap90], and database replication even more so. Database replication is an ongoing research domain, both in academia and industry.

This thesis discusses database replication, and in particular techniques that have been recently proposed that rely on some network abstractions, called group communications.

1.1 Database Replication

Database replication has been an area of research for more than twenty years: the first publications on the subject appeared in the early eighties [Tho79, Gif79, Sto79]. Since then, database replication has been an object of research. The core issue is that while the techniques that were proposed are correct, they have been shown to perform badly if the number of sites increases [GHOS96]. The main reason for this is that those replication protocols were designed to impose minimal changes in the database engine. This yields high synchronisation costs and a high number of deadlocks. Legacy and performance are the two main reasons for this "black-box" approach. A large amount of information is stored in databases and used everyday, so changing the architecture and migrating the data would imply a lot of work.

As databases become more and more important for enterprises, performance expectations for replicated database have risen [Jaj99]. New replication mechanisms have been searched for. One way to get increased performance for replicated databases is to relax to consistency rules: this approach is currently used by commercial database systems [Ora98, Inf98] and has also been proposed as the

main avenue of research [BBC+98]. This approach, called lazy replication, offers promising perspectives for disconnected computing and has the advantage of following the "black box" approach. The main issue with lazy replication is that it does not adress the problem of consistent, fault-tolerant replication. Recently, group communication-based replication has been proposed as an alternative to lazy replication [SR96, Alo97, PGS98]. Those techniques use high-level primitives to offer consistant, fault-tolerant database replication with acceptable performance, but imply a "white box" approach – the database system must be changed to rely on group communication abstractions.

The work unit of a database is the transaction. Transactions are sets of instructions that are executed on the database as one logical unit. Transaction processing is defined by the ACID rules: Atomicity ensures that either that the transaction executes completely, or not at all, Consistency ensures that the transactions brings the database to a legal state, Isolation ensures that the parallel execution of two transactions has no side-effect on either transaction, Durability ensures that once a transaction is commited, its effect last forever, even in the event of failures. In order to build consistent database replication, the ACID properties must be enforced.

1.2 Group Communications

Group communications are a set of high-level communication primitives and tools that are designed to help build replicated services. The basic idea is to address all replicas as one entity [HT93]. The notion of group communication originated from distributed operating systems [CZ85], and is used in systems like Amœba [KT91, KT94]. Group communications have been an object of ongoing research which resulted in both theoretical results and prototypes. Theoretical results include precisely defined semantics and minimal conditions needed to solve certain problems [FLP85, CT96, CHT96]. Prototypes moved the group communication infrastructure out of the operating system, to become autonomous toolkits.

The Isis group communication system [BAD+84, BSS91] proved that group communication could be used in practical settings. Many other group communication toolkits were developed [PSWL94, MFSW95, vBM96, MMSA+96, DM96, BCH+00, HS00, BCH+00, MPR01], mostly in academic settings. With the emergence of middleware systems, the trend has been to integrate group communication toolkits inside the middleware infrastructure [WAL97, Fel98, BCH+98, SW99, GNSY00, OMG01].

All group communication toolkits offer similar high-level abstractions. Group communication mechanisms ensure two kinds of properties: agreement properties and ordering properties. Agreement properties ensure that all members agree on some value, or on the delivery of some message. Ordering properties ensure that messages are delivered to all members in certain order.

1.3 About this Research

This thesis was started in the context of the DRAGON project (**D**atabase **R**eplication b**A**sed on **G**roup Communicati**ON**) – a joint project between the Swiss Federal Institute of Technology in Lausanne (EPFL) and the Swiss Federal Institute of Technology in Zürich (ETHZ). The goal of DRAGON was to explore the possible use of the group communication techniques in the context of database replication.

One important aspect of this project was its inter-disciplinary nature. The database and the distributed system communities are very different. In order to build efficient replication techniques, a good understanding of both communities is needed. While the idea of using group communications for database replication is quite old – Chang proposed in 1984 to use broadcast protocols (a form a view synchronous broadcast) to simplify the design of a two phase commit protocol[Cha84] – this area of research has only become active again recently.

1.3.1 Research Objectives

A lot of research has been done in the domain of group communication-based database replication [SR96, PG97, PGS97, Alo97, PGS98, KA98, KA99, PGS99, KPAS99, HAA99b, PF00, KA00a, KA00b, PMJPA01, FP01, RJP01], and very promising replication techniques have been proposed. Yet research in this domain was mostly explorative: new techniques were proposed and their performance assessed, often in special and restricted settings.

This thesis explores database replication based on group communication in a systematic way. The goal is to get the "big picture" and to understand all issues related to the problem. This is done in three ways: classification of replication techniques, definition of precise failure semantics, and performance evaluation. All three approaches require a solid groundwork to understand the differences and common points between the models of database systems and group communication systems.

1.3.2 Research Contributions

The research contributions of this thesis match the research objectives of the project: classification of techniques, understanding of failure semantics and performance evaluation.

1.3.2.A Classification of Replication Techniques

A large variety of replication techniques have been proposed, yet the relationship between those techniques is often not clear. Similar techniques might appear very

different because they are defined using different models and terminologies. At the same time, subtle changes in one protocol can result in very different replication techniques. To better understand database replication and the different techniques proposed in the literature, a systematic classification is proposed.

A classification has two advantages: first it helps in understanding the concepts and the mechanisms underlying replication, and the requirements of replication schemes; second it helps to assert if all possible replication schemes have been described: while many replication techniques have been proposed, some combination might have been overlooked. By exploring the solution space in a systematic way, such oversight can be spotted.

1.3.2.B Analysis of fault-tolerance semantics

One of the main goals of replication is fault-tolerance: the replicated system is expected to withstand some failure scenarios. Group communication systems and database systems have very different failure assumptions, so the guarantees offered by a system built using those two technologies are not trivial. Fault-tolerance is an end-to-end property: when multiple components are combined, the resulting system typically exhibits the fault-tolerance of the weakest component. Because of this, careful analysis is needed to understand the fault-tolerance of a system built by combining group communication and database elements.

We show what failure guarantees can be expected when linking together group communication and database. We also introduce new failure semantics that are better adapted to express the fault-tolerance of group communication-based database replication. We also propose new variants of existing replication schemes that maximise performance while respecting the failure semantics.

1.3.2.C Performance Evaluation

In order to understand the performance and the behaviour of different replication schemes, those replication schemes were implemented in a simulator and their performance measured. Classification gives a qualitative view of the replication techniques, performance evaluation gives a quantitative view.

The performance evaluation uses the classification as a basis: techniques from all the relevant categories are compared. One important contribution of the performance evaluation is the performance of some promising group communication-based techniques the performance of which was never evaluated in comparison to other techniques. The performance of the variants proposed in the analysis of fault-tolerant semantics is also evaluated. Other optimisations of group communication-based replication techniques are also presented and evaluated.

1.3.3 Thesis Organisation

The thesis is organised as follows: Chapter 2 discusses system models, abstractions and practical issues for both the database and distributed system communities and

gives an unified model. Chapter 3 presents a classification of database replication techniques. Chapter 4 discusses recovery and fault-tolerance issues. Chapter 5 presents a performance evaluation of different replication techniques. Finally, Chapter 6 summarises the major results of this work and outlines future research directions.

Chapter 2

Database and Group Communication Models

Verbosity leads to unclear, inarticulate things.

Dan Quayle

Using group communication systems to build a replicated database might seem a logical choice and has been proposed for some time [Cha84]. By leveraging group communications toolkits, the design of a replicated database would be easier and would permit a layered design where network functionality is separated from the database application (Figure 2.1).

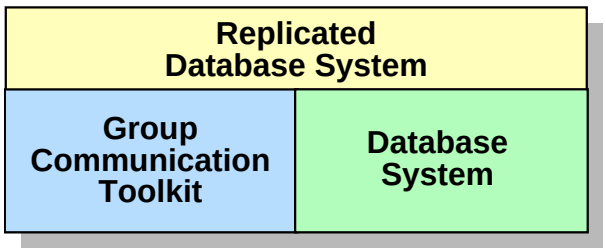

Figure 2.1: Basic building blocks

While this approach seems logical and practical, it implies assembling two components originating from two very different communities: the distributed system community, and the database community. Because of this, both components rely on different models and assumptions and offer very different properties.

Each community tends to concentrate on certain problems and simplify issues outside its scope. The database community tends to have a simple network model,

and the distributed system community a very simple processing model. Both communities have different goals, priorities, but also different models and metrics.

To build an efficient system by combining group communications and databases, both models must be understood and somehow reconciliated. The resulting unified model can help us describe and design replication strategies in a synthetic ways. This means grouping all communication primitives in one communication module, regardless of the fact that they originated in one community or the other.

This chapter describes both sides of the "fence" and presents the philosophies and models of both communities. It then presents an unified model that will used in the following chapters. This chapter is structured as follows: Section 2.1 gives an overview of both communities, Section 2.2 presents the distributed system model and associated issues, Section 2.3 presents the database model, Section 2.4 presents practical issues that further distinguish both communities. Section 2.5 gives a summary of the differences and presents the unified model.

2.1 Overview

While replication has been considered by both the database community and the distributed system community for quite some time, the motivations for replication are quite different. The focus of each community is also quite different. This section present the basic goals and philosophies of both communities.

2.1.1 Distributed Systems

The distributed system community focuses on systems that are distributed on a network. The components of the system are distributed on different computing nodes and communicate by exchanging messages. The main goal of the distributed system community is to make the system consistent by offering some guarantees on the message flow and ordering. This is achieved by adding an abstract layer on top of the actual network – in the same way the operating system adds an abstraction layer on top of the actual hardware. This architecture is illustrated in Figure 2.2. Different ordering an consistency levels have been specified and the corresponding algorithms implemented.

The system built on top of the group communication infrastructure can be an application, or another level of infrastructure, for instance a virtual shared memory environment [BKT92].

The main goal of replication in the distributed system community is fault tolerance. A system can be replicated by having multiple copies: if one copy fails, work continues on other copies. The main problem is to keep all copies synchronised and to hide failures as they occur. Ideally, failures are handled in a way that is transparent for the application.

Because the communication infrastructure only concerns itself with the ordering of messages and their reliable delivery, the replication schemes associated with

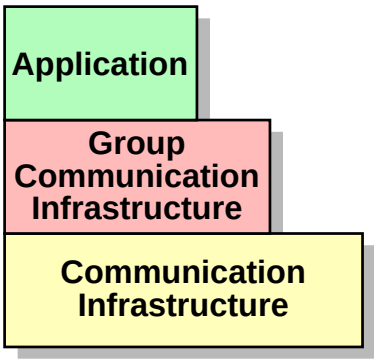

Figure 2.2: Group communication stack

those communication primitives tend to consider a simple application model. Issues like multi-threading, non-determinism, load balancing and large state transfers are typical of complex applications. Those issues are often considered out of the scope of group communications, and therefore not addressed in the models considered. This means that in order to replicate a complex application, those issues have to be handled by the application – therefore an application cannot simply be replicated by using the adequate communication primitives, a redesign is needed.

Another problem is that the communication primitives are often lacking the precise semantics the applications would need; some applications only need certain messages to be ordered, so ordering them all might be a waste of resources. Also group communication can be difficult to integrate with communication channels that are not under the control of the group communication system [CS93]. New communication primitives [PS99] and flexible group communication system [BCH+00, HS00, MPR01] have started to address those issues.

An important issue is the lack of focus on practical issues: a lot of research has been done to define precise models and understand their properties, but not much work has been devoted to comprehensive performance study or practical quantification of fault-tolerance [Kim00]. This has led to a wide variety of algorithms that where not classified until recently [Déf00] it was also recently that their relative performance was studied [UDS00]. Because of this background, group communication protocols tend to be perceived as slow and resource-consuming.

Those limitations are the main reason why group communication primitives have not been deployed much in actual systems and current work is mostly done in academic settings.

2.1.2 Database Systems

The database community focuses on systems that handle data and its storage. While resilience is an important issue in databases, replication is not a central focus of the database community, instead databases are expected to recover correctly

after a crash.

Replication has often been considered for administrative reasons (the data pertaining to certain location being in those location), performance reasons (moving data closer to users), and also fault-tolerance. Replication is often used at hardware level to ensure fault-tolerance using specialised hardware (RAID, disk shared between processors, etc.).

Database systems rely on the notion of transactions. Transaction execution satisfies the ACID properties (Atomicity, Consistency, Isolation and Durability, see Section 2.3.3). The system is not expected to hide failures: in case of a crash, transactions are aborted and the system rolled-back to a consistent state. The basic of today's database where established by Eswaran *et al.* [EGLT76].

Distributed databases have been considered for some time [LSG+79], and are usually handled as an additional layer on top of existing systems. Distributed transactions span one sub-transaction on each node, those sub-transactions are synchronised using an atomic commitment protocol that ensures atomicity between all nodes. Figure 2.3 illustrates this architecture.

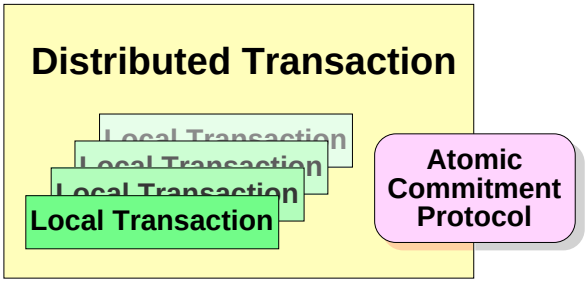

Figure 2.3: Distributed transactions

Replication is traditionally seen as a particular case of distributed databases, where one distributed transaction updates all n copies of the database. If one server crashes during transaction processing, the transaction aborts and needs to be restarted. The failure of one server is therefore not fully transparent. This is not so much an issue, as transaction typically abort for other reasons (for instance concurrency control) so users have to cope with transaction aborts anyway. This approach has been shown not to be scalable [GHOS96], and expensive performance-wise, therefore it is not often used and it was proposed that research concentrates on relaxed constraints [BBC+98]. This lack of transparency of failures and the unpredictable aborts make database systems unsuitable for critical and real-time applications [Kim00].

Database systems are intensively used in the core of enterprises, therefore there is a heavy emphasis on performance. Formal models are often relaxed to get better performance. For the same reason, database systems tend not to rely on underlying layers, like operating systems – instead the relevant functionality is implemented in an ad-hoc way inside the database system.

2.2 Distributed Systems Model

A large number of system models have been considered by the distributed system community (see [Gär01]) and describing them all is beyond the scope of this work. A model is needed to specify group communication primitives, but at the same time, the discussion about the use of group communication for database replication should not tie us to one specific model. Because of this, we will use an abstract model, and describe communication primitives and the problems they solve in this context. Then we will show how this abstract model relates to actual models. Vitenberg *et al.* have proposed an unifying framework for group communications systems in [VCKD99], but this framework concentrates on systems using views.

In this section, we give an overview of some classical models and their relationship. We do not present a new model, but rather a general framework that integrates different existing models. Not all existing models will be described in this section, but only those that are usually considered in the literature: the static crash no-recovery model [FLP85] (also called crash-fail), the dynamic crash no-recovery model [BJ87] (also called the crash fail model with views) and the crash-recovery model [ACT98]. They are conceptually close to each other and can be described in a synthetic way. Some models not considered here rely on randomisation [BO83], others, like the timed asynchronous model [CF99] are more complex and based on actual system measurement, but can be reduced to models presented here.

The different models considered are classified according to two basic criteria: time and state, i.e, how they consider time, and how the state of each process is handled. This section is structured as follow: Section 2.2.1 gives some basic definitions, Section 2.2.2 presents different timing hypothesis, and Section 2.2.3 presents different state handling policies.

2.2.1 Definitions

The system modelled is a group of *processes* that communicate using *messages*. Messages are transmitted through *channels*. No other mean of communication is available for processes to communicate, and processes do not have access to a global clock. The sending and receiving of messages are *events* that happen on the process receiving or sending the message.

In order to specify the communication protocols, we need to define what process take part in the protocol, and what processes do not. A crashed process that will never recover cannot be expected to deliver a message. We also need to define channels: again, a channel cannot be expected to deliver a message to a process that has crashed and never recovers. Different models have different definitions for processes and channels. This yields different specifications for each problem depending on the model, something we wish to avoid. Because of this, we need a more general abstraction.

2.2.1.A Processes

Figure 2.4 illustrates how we categorise processes. There are two general categories, processes that crash, and processes that eventually stay up long enough to finish the computation. Those two criteria are generally used to define processes in the group communications models we consider. If we consider the intersection, we end up with three categories:

1. processes that never crash
2. processes that crash, recover and are eventually up long enough, and
3. processes that are never up long enough.

We say a process is *green* if it never crashes. A *red* process is a process that behaves in a way that prevents it from participating in solving the problem, for instance by crashing and never recovering or by being *unstable*: the process crashes, recovers, crashes again, etc. An unstable process is never up long enough to do some useful work. A process might be neither *red* nor *green*: it sometimes crashes, but eventually stays stable for enough time to take part in the computation. We call those processes *yellow*.

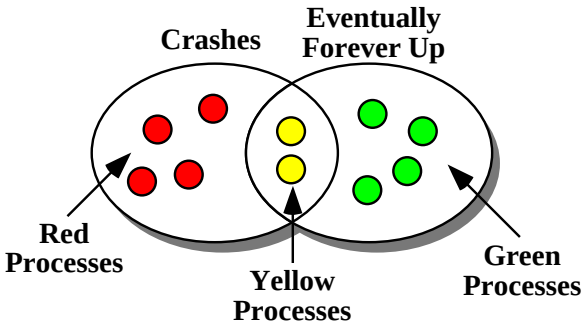

Figure 2.4: Process types

In the crash no-recovery model (dynamic or static) [FLP85, CT96], green processes are called *correct*, and never crash. Red processes are called *incorrect* and fail. As processes do not recover, the problem of unstable processes is not encountered. All processes are therefore red or green and there are no yellow processes.

In the crash recovery model [ACT98], a red process is one that crashes and never recover, or one that is unstable. Such processes are called *bad*. A *good* process is one that never crashes, or one that crashes, recover, but eventually stays up forever – this corresponds to a *non-red* process. A good process might be green or yellow.

The timing aspect of the model also has an impact on the specification of red and green processes. If there are no timing assumptions in the model, then “long

enough" means "forever". So if there are no timing assumptions, a green process stays up forever. If the timing model is synchronous, there are timing bounds on both process speed and message delivery time. Therefore, "long enough" means longer than some time T, where T is maximum bound on time needed to solve the problem considered. Timing aspects for the model are discussed in Section 2.2.2.

2.2.1.B Channels

The other important aspect of our abstract model are communication channels. Here again we have conflicting definitions, depending on the model. In order to have an abstract model, we need a specification that merges both models.

Crash no-recovery models tend to use either the *reliable* [BCBT96] or the *quasi-reliable* [ACT97] channel specification. The reliable specification states that if a process (even a red one) sends a messages to a green process, then this message will be eventually received. Reliable channels are not a reasonable assumptions, as real systems typically use buffers and pipelines. If a process sends a message and then crashes, those buffers will be lost. So expecting messages sent by crashing processes to be delivered is not very reasonable.

The quasi-reliable specification states that if a green process sends a message to another green process, then this message will be eventually received. While this makes sense in a model where process do not recover, this poses some problems in a model where processes can crash and recover – the quasi-reliable specification does not define how yellow processes should handle messages.

The crash recovery models allows yellow processes, that is processes that crash and recover. The fact that they crash does not prevent them from participating in the protocol: when they recover, yellow processes will be expected to take part in distributed algorithms and therefore to send and receive messages.

When a message reaches a process that is crashed, it will be lost. So we cannot expect yellow processes not to lose messages. On the other hand, if yellow processes can lose all messages, it will be impossible to solve any problem. For this reason, crash recovery models tend to rely on *fair-loss* channels. In this specification, the loss specification is different: if a process p sends an infinite number of messages to a process q and q is non-red, then q will receive a infinite number of messages from q.

The important thing to notice is that those two models are, in fact, not contradictory. Expecting no loss if both the sender and the receiver are green (so they never crash) does not contradict expecting a fair loss if either the sender or the receiver might crash. So we need to specify the loss of messages in two ways to cover both models. The resulting specification is the following:

No Creation If process q receives message m from p then p sent m to q.

No duplication Process q receives message m from p at most once.

Quasi No-Loss If p sends m to q and p, q are *green*, then q eventually receives m.

Fair Loss If p sends messages to a non-red process q an infinite number of times, then q receives an infinite number of messages from p.

Channels defined with this specification are equivalent to fair-loss channels in the crash recovery model and to quasi-reliable channels in the crash no recovery model. In the crash no recovery model, the *fair loss* clause can be reduced to the other clauses and does not apply, so the resulting specification is that of *quasi-reliable* channels. In the crash recovery model, the *quasi-no loss* clause can be reduced to the other clauses, so the resulting specification is that of *fair loss* channels.

Proposition 1 *The fair-loss property can be reduced to the other clauses in the crash no-recovery model.*

Proof 1 There are no yellow processes in this model, so p and q are either red or green. As q is non-red, q is green. If p is green, then the quasi no-loss clause applies, if p sends a message to q, q will receive the message. So if p sends an infinite number of messages, q will receive an infinite number of messages. If p is red, it will crash. Therefore it cannot send an infinite number of messages before crashing.[1]

Proposition 2 *The quasi no-loss property can be reduced to the other clauses in the crash recovery model.*

Proof 2 Processes p and q are green and will therefore never crash. If p sends message m an infinite number of times to q, m will be received. Therefore quasi no-loss can be implemented using fair-loss channels and the following algorithm: When p sends a message m to q, it repeatedly sends m to q until it receives an *ack* message from q. Each time q receives a message from p, it sends an *ack* message. As neither p nor q will crash, eventually, q will receive m.

2.2.2 Synchrony

One important aspect of models is the timing issue. If the model is *synchronous*, there is a bound δ on the relative processing speed of processes, and a bound Δ on the time needed for a message sent by one process to be delivered by another. This means that if a process executes one instruction step in one time unit, then all other process will execute it in a most δ time units. Also, a message sent at time t will be received at latest at time $t + \Delta$.

[1] We assume that a process cannot execute an infinite number of operations in a finite time.

On the other hand, if the model is *asynchronous* such bounds do not exist. There is no bound on the relative processing speed, or on message delivery. Because there is no time-bound, time has basically no meaning in the asynchronous model, so it is sometimes called a *timeless* model.

It has been shown [FLP85] that the asynchronous model is not powerful enough to solve the consensus problem if one or more process crash. Most group communication problems can be reduced to consensus. So if consensus cannot be solved, neither can those problems.

In order to be useful, the asynchronous model can be augmented with *failure detectors*. Failure detectors are oracles that tell one process about the state of others. Consensus can be solved in an asynchronous model using failure detectors [CT96]. Failure detectors can be perfect or imperfect – an imperfect failure detector might incorrectly suspect a non-failed process – this is called a false suspicion. A failure detector should detect red (crashed) processes (completeness) and not suspect non-red (non-crashed) processes (accuracy). Another kind of failure detector have a leader election property, i.e all processes agree on one non-red process[Lam98]. We consider the following failure detectors:

P This failure detector is perfect, crashed processes are eventually suspected by all processes (strong completeness), and no non-crashed process is ever suspected (strong accuracy).

$\diamond S$ This failure detector satisfies weak accuracy (non crashed processes might be suspected, but eventually, all non crashed processes will not be suspected) and implements strong completeness.

Ω This failure detector satisfies eventual leader election. That is, eventually a leader is selected by all processes.

$\diamond S_u$ This failure detector is similar to $\diamond S$ but is defined in the crash-recovery model [ACT00] this means that it can handle *unstable* processes. Strong completeness therefore also requires all processes to suspect unstable processes.

Failure detectors Ω and $\diamond S$ are equivalent, in the sense that both can solve consensus and that there is an algorithm to transform each failure detector into the other [CHT96].

2.2.3 Process State

The other important aspect of a model concerns the state of the different processes. The state of each process is split into different layers, corresponding to the conceptual layers used in the context of group communications: application, group-communication, communication infrastructure (see Figure 2.5). During the normal course of operation, each level maintains its own state for processing.

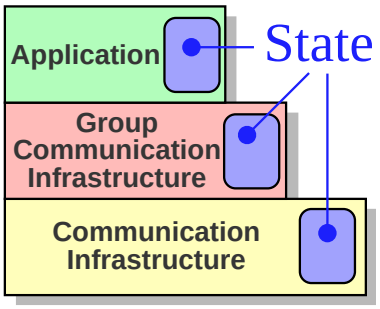

Figure 2.5: Group communication stack state

In case of a crash, part of the process state might be lost, because this part was only stored in the volatile memory of the process. The state might be preserved if replicated in the volatile memory of other processes or stored in stable storage. Different models handle this state information in different ways. What part of the process state is preserved in case of crash will decide if processes will be able to recover and continue processing. This will in turn give each model certain properties. We describe and classify models according to how they handle state, which makes it possible to establish a relationship between different models and describe them in a general way.

Communication Infrastructure Layer. The lowest layer of each process state concerns the communication layer. Because this layer is traditionally handled by the operating system and the networking hardware, its state is usually volatile and cannot survive a crash. That is, this state is not stored on stable storage, nor replicated. Storing messages in stable storage would be very expensive in terms of performance, as stable storage is usually implemented using slow disk technology. It would also require a redesign of the operating system and the communication hardware. Replicating the state does not make sense: at this level, while some group abstraction exist (like IP-multicast), most abstractions are peer-to-peer relationships. This is why we consider the channel model described in 2.2.1.B – messages are transmitted as long as there is no crash. If there is a crash, all state, including pending messages, might be lost.

Application Layer. The highest layer is the application layer. The state of the application is what needs to be replicated. When a process recovers, the application layer state must somehow be reconstructed. This can be done in two ways: either by getting the state from another process, or by replaying the messages to rebuild the state. In the first case, this means that the group communication layer can access the state from the application layer of another process. In the second case, this means that the communication layer can replay all messages. This is only possible if the communication layer stored those messages. This distinction has

some impact on the model and will be discussed, but will be discussed in Chapter 4.

Group Communication Layer. This intermediate layer offers advanced communication primitives: it gives applications the tools and primitives needed by the overall system both for normal operation and recovery. Those primitives provide the groundwork for replicating the application layer.

The communication infrastructure is typically given by the operational settings, and can therefore not be adapted for replication. The application layer should be isolated from the model. By using the abstractions given by the group communication layer, design and validation are simplified and implementations are portable. An application simply uses group communication primitives and is built on the properties of those primitives. The design of the application layer should not be affected by timing considerations, failures detectors issues, etc. The role of the group communication layer is to offer high level primitives with their own guarantees, and enforce those guarantees in regard of the model. For this reason, model considerations affect mostly the group communication layer. The next section describe how the group communication layer handles state information in different models introduced in Section 2.2.1.

2.2.3.A Static Crash No Recovery Model

In this model, processes do not recover once they crashed. Because of this, recovery is not an issue. Therefore, state does not need to be replicated for recovery purposes. The application state will probably be replicated (like, for instance, for state machine replication [Sch90]), but the group communication system will not need that the application layer gives out its state for recovery purposes, as there is no recovery. As processes do not recover, there is also no point in restoring the state of the group communication layer. While this model has been the basis for theoretical research [FLP85, CT96], it is not very useful for practical cases.

2.2.3.B Dynamic Crash No Recovery Model

The dynamic crash no recovery model has been introduced in the Isis system [BJ87]. It addresses the shortcomings of the static crash no recovery model. The key idea of this model is to allow a dynamic number of process in the system. The composition of the group at some point in time is called the *view*. A view v_i is a set of processes $v_i = \{p_{i1} \dots p_{in}\}$. When a process crashes, or is suspected of crashing, it is excluded from the view. New processes can join the view. While a process cannot formally recover, it can rejoin the group once it recovers with a new identity – this is called a new *incarnation* of the process. The system is defined as a sequence of views $v_0, \dots v_n$, a new view being installed each time a process leaves or joins the view – this event is called a *view change*. Between two view changes, this model behaves like the static crash no-recovery model. The view information is replicated in the group communication state of each process. The mechanisms

to achieve this replication are those of the group communication layer itself. The notion of views is what makes this model very elegant: the information regarding replication is itself replicated. Also views address two issues: the dynamic composition of the group (which is an administrative issue) and the handling of crashed processes (which is a fault-tolerance issue).

Because only the view information is replicated in the group communication layer, all recovery actions rely on it. In order to become part of the group, a recovering server must therefore update the view to include itself in the view. If the view is not accessible (because of too many process failures), recovery is impossible, and the system blocks.

2.2.3.C Static Crash-Recovery Model (no stable storage)

In the crash-recovery model, a recovering process can take part in a computation immediately – there is not need to wait for a view change to be installed. This means a process can take part in a protocol that has started while the process was crashed. In order to do this, part of the state of group communication system must be restored to the recovering process.

For instance, if we consider a consensus protocol, a certain value v can be decided while process p was crashed. This means that v must be kept as long as p has not decided. If multiple instances of consensus where run, all values v_1, v_2, must be kept. If we consider total order broadcast, all messages that have been delivered while p was crashed must be kept to be sent to p when it recovers. So part of the state of the group communication layer must be replicated on all servers.

When a process recovers, the application layer state can be reconstructed either by querying the state from another process (like in the dynamic crash no-recovery model) or by using the state information of the group communication layer, i.e the communication layer replays messages to the application layer. The up-to-date state is obtained by replaying messages to an old state (or even the initial state) [RR00]. This means that a recovery process can obtain an up-to-date state, even if no application layer has the most up-to-date state.

	static membership	**dynamic membership**
crash no-recovery	static crash no-recovery	dynamic crash no-recovery
crash recovery	static crash recovery	dynamic crash recovery

Table 2.1: Crash and membership properties of different models

One important thing to note is that the crash recovery model usually considered is static [ACT00] – the set of processes considered never changes. So while this model addresses failures, it does not handle system reconfigurations. Table 2.2.3.C shows all the combination of group membership and failure model: group membership can be either static or dynamic, crash behaviour can be either crash recovery, or crash no-recovery. A dynamic crash-recovery model would be possible, but

quite complicated: it would imply a model that handles both the view mechanism for long term administration and the crash-recovery model to handle short-term failures. No such model has ever been described in the literature. Some research about separating administrative issues from fault-tolerance issues by using different failure detections mechanisms are discussed in [CBDS01].

2.2.3.D Static Crash-Recovery Model (with stable storage)

This model is similar to the crash-recovery with no stable storage, the main difference is that each process stores the state of the group communication layer in stable storage. When a process recovers, the state of its group communication layer can be reconstructed either by requesting it from other processes (like in the case when no stable storage is present), or by reading it from stable storage. Stable storage in itself might not be sufficient, because the state in stable storage only does not always contain the state related to the most recent iterations of the protocols.

The use of stable storage makes it possible for the group to survive even catastrophic failures, i.e, the crash of all processes. Without any stable storage, recovering from such a failure is impossible: if state information is not stored on stable storage, it is definitely lost if a catastrophic crash occurs.

One important thing to note is that the presence or the absence of stable storage only concerns the group communication layer. The application layer might have access to stable storage and use it for its own purposes (this would typically be the case for a replicated database). The properties obtained by relying on the stable storage of the application layer are discussed in Chapter 4.

2.2.3.E Summary

Table 2.2 summarises how the different models handle the state of the group communication layer.

Model Name	**Group Communication State**
Crash No Recovery	Not replicated
Dynamic Crash No Recovery	View information replicated
Static Crash-Recovery (no stable storage)	Replicated
Static Crash-Recovery (with stable storage)	Replicated & stored in stable storage

Table 2.2: State handling of different models

2.2.4 Communication Primitives and Associated Problems

This section describes some problems that the group communication system solves. Group communications are generally presented in the crash no recovery model [HT93]. Here, they are specified for our general model. Note that we also consider atomic commitment as a group communication primitive, which is not traditionally the case. Atomic commitment originated in the database community and is often not considered as a group communication primitive. Yet atomic commitment is clearly an *agreement problem* that can be reduced to consensus [Gue95]. Atomic commitment protocol is a useful primitive to implement actual systems and the implementation of atomic commitment protocols can be simplified by using group communication tools and techniques [Cha84, BT93, JPPMAA01]. So integrating atomic commitment into a group communication toolkit makes a lot of sense. In our case, it makes it possible to group all communication problems in one module, without encumbering the database model with communication issues.

The primitives that are described in this Section are typically "strong", in the sense that their guarantees are absolute (not best effort). They also all implement the uniform specification of the problem, so typically, processes that crash are not allowed to do "bad" things, like delivering messages in the wrong order, or delivering a wrong decision. Primitives that solve weaker versions of those problems are presented in Section 2.4.6.

Because the problems are specified in our general model, some properties only apply to green, red or non green processes. In general the properties are uniform, and apply to all non-red processes (i.e. processes that are eventually forever up). Typically, this means that all non-red process need to deliver a certain message. In the cases where properties apply to other classes of processes (for instance only green processes), the issue will be discussed.

2.2.4.A Uniform Reliable Broadcast

Reliable broadcast is a primitive that ensures that all processes in a set get a message even in the case of failure. Reliable broadcast defines two primitives R-broadcast(m) and R-deliver(m), specified as follows:

Validity If a process R-delivers m then it was R-broadcast by some process.

Uniform Agreement If a non red process R-delivers a message m, then all non-red processes eventually R-deliver m.

Uniform Integrity For every message m, every process R-delivers m at most once, and only if it was previously R-broadcast by sender(m).

Uniform reliable broadcast can be solved if there is a least one green process [CT96]. In the crash-recovery model with stable storage, uniform reliable broadcast can be solved if there is at least one yellow process.

2.2.4.B Uniform FIFO Reliable Broadcast

FIFO reliable broadcast is similar to Reliable broadcast, but enforces a First-In First Out policy. This policy only concerns messages from the same source. Formally, FIFO reliable broadcast follows the definition of Reliable Broadcast augmented by the following primitive.

FIFO If a process q RF-delivers m_1 before m_2 and $p = sender(m_1) = sender(m_2)$ then p sent m_1 before m_2.

FIFO reliable broadcast is trivial to implement using sequence numbers.

2.2.4.C Uniform Consensus

We consider that consensus is a core problem of group communications. Many agreement problems can be reduced to consensus. That is, algorithms that solve those problems can be built if an algorithm that solves consensus is available. Intuitively, the problem of consensus is to have all processes agree on one same value. Consensus is defined by two primitives: propose(v) and decide(v). Consensus is specified as follows:

Uniform Validity If a process decides v, then v was proposed by some process.

Uniform Agreement No two processes decide differently.

Uniform Integrity Every process decides at most once.

Termination Every non-red process eventually decides.

Consensus can be solved in the synchronous model or in a asynchronous model with a $\diamond P$ failure detector if there is at least one green process. In the asynchronous model, consensus can be solved using the with the $\diamond S$ [CT96], the $\diamond S_u$ [ACT00] and the Ω failure detector [Lam89], as long as $green > red$.[2] If stable storage is present, consensus can be solved if $green + yellow > red$ [ACT00].

2.2.4.D Uniform Total Order Broadcast

Total order broadcast is a communication primitive that ensures that all processes deliver the same messages in the same order. This primitive is also often called atomic broadcast. Atomic broadcast defines two primitives: A-Broadcast and A-deliver. It is defined by the same properties as reliable broadcast, with the following additional property:

Uniform Total Order If two process p and q A-deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'.

[2]The condition $green > red$ is sufficient regardless of the fact that the model permits yellow processes or not.

Total order broadcast can be reduced to the consensus problem [CT96]. That is, one can implement total order broadcast using consensus. For this reason, total order broadcast can be solved in the same situations that consensus. Total order broadcast can be augmented with the FIFO property, yielding FIFO total order broadcast.

2.2.4.E Uniform Non-blocking Atomic Commitment

Intuitively, this protocol ensures that if all processes vote *yes*, then the outcome is *yes*. If not, the outcome is *no*. Atomic commitment is a problem that originated in the database community, and is usually defined only with the first two properties (uniform agreement and uniform validity), when only defined with those two properties, it is called *blocking* atomic commitment. While the properties required are always uniform, this protocol is often not called uniform, as the non-uniform specification is never considered. Uniform non-blocking atomic commitment defines two primitives: AC-vote(v) and AC-decide(v), where $v \in \{yes, no\}$. Non-blocking atomic commitment ensures the following properties:

Uniform Agreement No two processes decide differently.

Uniform Validity If a process AC-decide *yes* then all process have AC-voted *yes*.

Fair non-Triviality If all processes are green, and they all AC-vote *yes* an infinite number of times, then *yes* will be AC-decided an infinite number of times.

Termination Every non-red process eventually AC-decides.

Intuitively, the non-triviality property states that if nothing goes wrong, and all sites vote *yes* all processes should decide *yes*. Note that this property is defined on *green* processes. The most logical specification for non triviality would be the following:

Strong non-Triviality If all processes AC-vote *yes* and all processes are green, then all processes eventually AC-decide *yes*.

The main issue with this specification is that the problem cannot be implemented using imperfect failure detectors. The following specification can be implemented using failure detectors [Gue95] (by reducing it to consensus):

Weak Non-triviality If all processes AC-vote *yes* and all processes are green, and none is suspected by a failure detector then all process eventually AC-decide *yes*.

The problem is, this introduces failure detectors in the specification (see Section 2.2.2). This is why we choose fair non-triviality. If the failure detector guarantees weak accuracy (like $\diamond P$) then the weak non-triviality enforces the fair non-triviality property.

Proposition 3 *If the model defines a failure detector that guarantees weak accuracy, then the weak triviality property implements the fair non-triviality property.*

Proof 3 If all processes are green, eventually they will not be suspected (weak accuracy). Once they are not suspected any more, if they all AC-vote *yes*, they will AC-decide *yes* (weak non-triviality). Therefore if they AC-vote *yes* an infinite number of times, they will AC-decide *yes* an infinite number of times.

2.3 Database Systems

In the previous section we have described the model for group communication. In this section we present the formal model for databases. While the previous section exclusively describes the group communication system model, this section does not only describe a single, non-replicated database model, but also some communication considerations – mostly replica consistency criteria and how transactions are distributed. All advanced communications primitives are described in Section 2.2.

The model is mainly about transactions and their correct execution. It is interesting to note that most properties are defined on data, which is quite different from group communication systems where the model defines properties on events.

2.3.1 Transactions

Transactions are sequences of data operations terminated by one control operation. Data operations are *read* and *write*, control operations are *commit* and *abort*. Operations inside a transaction have a certain order (noted $<_t$) and must be executed according to this order.

Formally, the database D is composed of data items $\{d_1 \dots d_n\}$. A transaction $t = \{op_t^1, op_t^2, \dots op_t^m\}$ of length m is defined as a partial order of its operations with respect to operator $<_t$:

1. $\forall i < m \; op_t^i \in \{read(d_k), write(d_k)\}$
2. $op_t^i <_t op_t^j$ iff $i < j$
3. $op_t^m = abort \mid op_t^m = commit$

2.3.2 Histories

Transaction executions inside databases are formalised thanks to the notion of histories [BHG87]. A history defines in what order a set of transactions T are executed in a database. Let $T = \{t_1, t_2, \dots, t_j\}$. a *complete history* H over T is a partial order of read and write operations. The partial ordering is expressed using the operator $<_H$ such that:

1. $H = \bigcup_{k=1}^{j} t_k$ (All transaction are part of the history).

2. $\bigcup_{k=1}^{j} <_{t_k} \subseteq <_H$ (All partial orders $<_{t_k}$ are compatible with the partial ordering $<_H$).

3. If two transactions contain operations that conflict (they touch the same data item d_k and one of them is a write) then the ordering of those operations is the same than the ordering of their respective transactions.
   For two operations $op_1(d_k)$ and $op_2(d_k)$ where $op_1(d_k) = write(d_k)$ and $op_2(d_k), \in \{read, write\}$, such as $op_1(d_k)$ was issued by $t_1 \in H$ and $op_2(d_k)$ issued by $t_2 \in H$ and $t_1 \neq t_2$ either $op_1(d_k) <_H op_2(d_k)$ or $op_2(d_k) <_H op_1(d_k)$.

A *history* is a prefix of a complete history. Given some history H, $C(H)$ is the history obtained by removing the operations of transactions that did not commit in H.

2.3.3 ACID properties

Histories specify how operations execute inside transactions, but do not specify how transactions interact with each other and the system. This is specified in the ACID properties [GR93]:

Atomicity. A transaction's changes to the state are atomic, either all happen or none happen.

Consistency. A transaction is a correct transformation of state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation. Even though transactions execute concurrently, it appears to each transaction, t, that others executed either before t or after t, but not both.

Durability. Once a transaction is successfully completed (it commits), its changes to the state survive failures.

Those definitions are rather broad, and need to be clarified in our context.

Atomicity. Atomicity ensures the "all or nothing" property, that is, all the changes of the transactions are performed, or none of them.

Consistency. Consistency ensures that the data inside the database stays consistent. Database consistency must respect integrity constraints. Integrity constraints are rules that define what state of the database is legal, and what transitions in this state are permitted. If a transaction would lead to a state of the database that does not respect those constraints, it is aborted. Consistency is typically defined at the level of the application running on top of the database.

Isolation. Transactions can be more or less isolated from each other, the ANSI standard [ANS92] defines therefore three isolation levels. We consider level three, which is called *serialisability* [GR93]. Serialisability is the strongest isolation level: it ensures that the effects of transactions executing in parallel are equivalent (in the sense of the effect of write, and the values read) to a serial execution. Other isolation levels include *snapshot isolation* which permits a transaction to read old data. Those weaker level of isolation are typically used to increase performance and will not be considered here. Isolation is handled by the concurrency control mechanism and is detailed in the next section.

Durability. Durability is based on the assumption that if the server crashes, it will eventually recover, and that critical data has been stored on stable storage [Had88].

The ACID properties must hold for committed transactions. If the system cannot guarantee the ACID properties for transaction t, it will abort t. Aborting a transaction is an unilateral decision. A transaction can be aborted either by an explicit instruction (abort) inside the transaction or by the database itself. If either party (the transaction or the database) decides to abort the transaction, the other party cannot prevent this abort.

Each of the four ACID properties can lead to the abort of a transaction. If not enough stable storage resources are available, the system might no be able to enforce atomicity or durability (there is not enough disk space to write all items of the transaction). The database enforces integrity constraints, which ensure that the database always stays consistent. Those integrity constraints are defined by the application using the database, and can be linked to the semantics of the information inside the database. For instance a certain account type cannot have a negative balance. If the result of a transaction t violates the consistency of the database, then t is aborted.

It is important to note that the ACID properties are safety properties. They ensure that nothing *wrong* happens to the data. They do not guarantee that the system will make any progress. Thus one could implement a database system that aborts all transactions. This system would be of no use, but would respect the ACID properties. This poses some problems when integrating a database system with algorithms coming from the distributed system community, where liveness properties are always specified.

2.3.4 Concurrency Control

Isolation can be enforced in a simple way in a database – if all transactions are executed sequentially, isolation is enforced and no transaction needs to be aborted. This concurrency control policy is very inefficient, so more reasonable concurrency control techniques try to process multiple transactions in parallel while enforcing isolation for each of them. The two main approaches to concurrency control are

called optimistic and pessimistic concurrency control and are described below. Actual concurrency control system are usually not strictly optimistic or pessimistic, they are somewhere between both extremes.

Two operations op_1 and op_2 conflict if they are part of different transactions, access the same object, and either op_1 or op_2 is a write operation. Two transactions t_1 and t_2 conflict if they contain operations that conflict, i.e, $op_1 \in t_1$, $op_2 \in t_2$, op_1 and op_2 conflict (see Section 2.3.2). The concurrency control mechanism handles conflicting operations so that conflicting transactions appear to be executed serially (in terms of the results of read and writes).

2.3.4.A Pessimistic Concurrency Control

Pessimistic concurrency control is done using locks. Access to each item of the database is controlled by a lock. Before a transaction accesses an object, it has to acquire its locks. Two transactions can share a lock if they do not conflict. Usually, locking is done using the two-phase locking protocol (2PL): during the first phase the transaction acquires locks, the second phase starts when all locks are acquired, which occurs after the last operation (read or write). During the second phase, locks are released.

A variant often used is strict 2PL. In this case, write locks are kept until the transaction commits or aborts. Two-phase locking guarantees isolation, additionally, strict two-phase locking avoids cascading aborts. Cascading aborts happen when the abort of a transaction t_1 forces the database to abort t_2 for concurrency control reasons (in this case, because it acquired a lock that the aborting transaction released). In turn, aborting t_2 might force the database to abort t_3, etc.

Two-phase locking can lead to deadlocks. Such deadlocks have to be detected, either by building the *wait-for* graph and finding cycles in it, or by using time-outs. In case of a conflict, one of the transactions is aborted. Deadlocks can be avoided if the whole transaction is known in advance, which is the case for *stored procedures* (see Section 2.4.4). While deadlock detection based on time-out is non-deterministic, wait-for-graph-based deadlock detection can be deterministic.

2.3.4.B Optimistic Concurrency Control

Optimistic concurrency control does not use locks: transactions are executed concurrently. At commit time, they are certified. Certification consists in checking if the transaction's execution respects the serialisability property. If serialisability cannot be ensured, for instance because transaction t reads a stale version of an object, t is aborted.

Optimistic concurrency control is considered more efficient in low conflict rate situations, because it requires less book-keeping and permits higher concurrency. In high conflict situations, the abort rates tends to rise, making this concurrency control policy unsuitable.

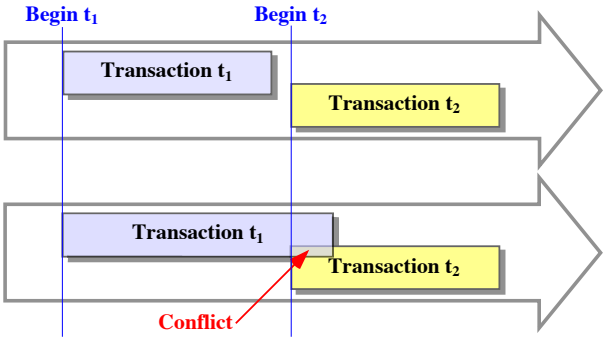

Figure 2.6: Optimistic concurrency control – non-deterministic scenario

Another drawback of optimistic concurrency control is the fact that it is not deterministic [WPS99]. Even if transactions are submitted in the same order, transactions might abort in some cases and not in others. Figure 2.6 illustrates the problem: consider two transactions t_1 and t_2, t_1 and t_2 conflict, t_1 is started before t_2. If t_1 terminates before t_2 is started, the execution will be serial and the concurrency control has no reason to abort either transaction. If t_2 starts before t_1 terminates, the concurrency control system will need to handle the conflict, for instance by aborting t_2. Therefore, depending on the timing of the system, transaction t_2 might commit or abort. Thus the system is non-deterministic.

2.3.5 Network Model

There is no precise formal model of the network used in database system. Communications are usually considered point to point with some synchrony assumptions. The crash of a process can be detected with a time-out mechanism, false suspicions (a process failing to respond before the time-out while not crashed) are not considered. This corresponds to the synchronous model, where there are known bounds to both message transmission delays and the relative speed of processes (Section 2.2.2).

The process model is similar to the crash-recovery model with stable storage (Section 2.2.3.D). Processes are assumed to crash and recover. The log of the database serves as stable storage. The database and the networking architecture share the same stable storage component (the log).

2.3.6 Distributed Transactions

Distributed transaction processing relies on two protocols, but many solution and replication techniques rely on different ad-hoc mechanisms. The two basic protocols generally used are a broadcast primitive and an atomic commitment protocol. The broadcast primitive is used to distribute the transaction to all processes, while

the atomic commitment protocol is used to decide on the transaction's outcome.

The atomic commitment protocol follows roughly the specification described in Section 2.2.4.E. The most common protocol used is the two-phase commit protocol (2PC) [GR93], which does not ensure the termination property – the 2PC algorithm might never terminate. Such an algorithm is called blocking.

Three-Phase Commit protocol (3PC) [Ske81] is non-blocking, but rarely used for performance reasons. There are many variants of the 2PC protocol, optimised for different cases [CSAH98]. One variant of 2PC merges the broadcast with the atomic commitment protocol: the resulting protocol does both the broadcast and the atomic commitment. New variants of the 2PC protocol replicate some components of the protocol in order to become non-blocking [JPPMAA01].

We consider *strict replica control* replication: the correctness criterion is *one-copy equivalence*. This criterion requires all copies of the database to be mutually consistent at the end of each update transaction. That is, all local histories have the same partial order $<_H$. Because all copies enforce serialisability, the resulting system enforces *one-copy serialisability*.

2.4 Practical Issues

It is quite possible to design a system that is consistent and efficient in a model, but is not practical to build because of practical issues. Models offer a rather abstract view of the system, but do not express certain things like what is "reasonable" and what is not. Database systems are advanced and complex systems, which cannot be reduced to a simple model: their design is also dictated by issues like deployment, compatibility, standard compliance and of course, performance. Understanding those practical issues is important, because they constrain the overall architecture. Some architecture that are logical in the model will not be considered for practical reasons. To some extent, the same consideration hold for group communication systems: the primitives specified in Section 2.2.4 are sometimes implemented in a weaker form, that does not offer the same guarantees, but better performance.

In this work, the problem of practical issues is further complicated because we are working with model originating from different communities. Each community has its own "hidden" assumptions about what is reasonable or possible – and in some case, those assumptions conflict. Finally, in some cases, these practical issues have a large impact on some replication techniques. This means that the same replication technique can have very different characteristics depending on the community. In this case, understanding the practical issues makes it possible to understand the difference between those two variants. This section describes those issues and discusses their implications.

2.4.1 Client Code

Group communication replication protocols tend to have a peer-to-peer approach. The system is represented as n equal processes. The model does not represent the capacities of processes (processing power, memory etc.). Processes can act as client, servers, or both. Any process can be a "server", as soon as an application is running inside this process. So it is common to have the clients use group communication possibilities. This makes it possible to handle failures transparently, as the client communicates with the whole group: the failure of one server can be hidden.

While group communication tend to consider a peer-to-peer architecture, databases usually rely on the client-server paradigm. Server and client machines tend to have different characteristics and fixed roles. Database applications run on dedicated machines, often with special hardware to minimise crashes and down-time: RAID disks, UPS power supply. Clients are very diverse: they can be interactive user terminals, application code running on a PC, or even a middle tier for more sophisticated applications. Client machines can crash or be disconnected very often. Client processes can have little or no local stable storage and little processing power. The number of client machines and processes is often an order of magnitude greater than server machines. Client machines are also often distributed geographically and in different administrative domains. This means clients also have limited access to both the database and the administrative structure (i.e., the replication scheme) for access control and security reasons.

Because of those distribution and heterogeneity issues, the assumptions on client must be kept simple. The large number of client machines makes deploying new protocols and interfaces a very expensive task. We will therefore consider a model where the client is not aware of an advanced replication protocol. Configuration information stored on clients is minimal: clients know how to connect to a static set of servers. This information is static and updated by human management. Clients connect to one server that will act as a proxy for the client. We call this proxy the *delegate* server. The interaction between the client and the delegate is done using standard mechanisms (SQL, stored-procedures). The whole burden of replication is therefore shifted to the servers. This enables us to different protocols and architectures for server-server connections and client-server connections. Among themselves, servers can use peer-to-peer and group communication techniques, as all servers reside on machines with roughly the same capacities. The client-server connection on the other hand, relies on standard protocols and middleware.

2.4.2 High-Level Transactions

Communication between the client and the server is often done using a high-level language like SQL. The request is parsed by the server and translated into low-level instructions that will be executed on the actual database. This translation requires access to the meta-data of the database that often is quite large (even larger that the

data itself). It also includes an optimisation phase that tries to find the minimal set of low-level instructions needed to execute the high-language command. This optimisation is an *NP-hard* problem, and is approximated by heuristics: it consumes a lot of processing power.

We call a *high-level transaction*, the transaction that is submitted from the client to the server. We call a *low-level transaction* the sequence of read and write operations described in Section 2.3.1. While formally there is no difference between high-level and low-level transactions, some replication techniques can more easily handle one or the other. Techniques where a high-level transaction is transmitted to all replicas will be called *high-level replication*: each replica parses and transforms the high-level transaction into a low-level transaction. *Low-level replication* techniques are those where the transformation is only done on the *delegate* server, and the low-level transaction is transmitted to all other processes.

High-level replication has the advantage or requiring little or no modification in the database system [PF00]. Also the high-level transaction can be orders of magnitude shorter than the equivalent low-level transaction: a single query can imply millions of read operations. This means that, if bandwidth is scarce, high-level replication can be an advantage. On the other hand, low-level replication means doing the costly transformation only once and therefore give better performance [Kem00]. Most techniques considered are low-level, if a technique is a high-level technique, we will mention it explicitly.

2.4.3 Cold standby vs Hot Standby

Formally, the state of a database system can be expressed either as the current state D_i or as the initial state D_0 and all the modifications $\delta_1 \ldots \delta_i$ that where applied to D_0. This means that it is equivalent to have (1) a database server that is up-to-date, or (2) the database server with a stale state and all the updates that change this state.

In practice applying all those changes can take quite a lot of time, typically if a back-up needs to apply all modifications $\delta_1 \ldots \delta_i$ before taking over, the system might well be unavailable during the time those updates are applied. Also keeping all those update consumes a large amount of memory, if the load is to high, then the server might even crash. For this reason, the database community makes the distinction between two types of strategies: *cold* and *hot* standby [GR93].

A *cold standby* server is a server whose state is not up-to-date, the server has only a list of changes. In order to be up-to-date, and be able to accept transactions, all those changes must be applied. A *hot standby* server has all the changes applied immediately, and can therefore accept requests without delay. The difference is especially important for replication schemes where one server acts as primary and the others as backup, but also in the case where one server is recovering, and is therefore acting as a back-up until it is up-to-date.

An important issue between cold standby and hot standby is flow-control: implementing cold standby is easy: the changes must simply be logged on stable storage. Installing those changes takes more time, as the transactions need to be

executed and in a precise order. This means that in a case of hot standby, some flow control mechanism might be needed to ensure the backup is not "overwhelmed" by new changes. This means that additional messages or communication rounds might be needed to be sure that a given server stays "hot". Even in cold standby situations, flow control might be needed to avoid that a slow backup is overloaded. Algorithms that formally do not need any synchronisation rounds to ensure correctness, might in practice implement such rounds to enforce some flow-control.

2.4.4 Interactive Transactions vs Stored Procedures

We distinguish between two basic transaction types. The first type, *interactive transactions*, are typically described in a high-level command language, like SQL. Such transactions are either issued by a human operator, or by a program running on a client machine that issues SQL commands. In both cases, the control of the transaction is not located on the server, but on a client (either in a client program, or handled by a human operator). In such transactions, the system does not know at the beginning of the transaction what operations it will contain and what data items will be touched. We call such transactions *dynamic transactions*.

The second type consists of *stored procedures*. Those transactions are invoked by a remote procedure call (RPC) mechanism. The transaction that will be invoked are represented in the form of code stored and executed on the server. Stored procedures are typically written by experts, and are used by application developers. Stored procedures offer better performance, access control, and simplified application development. They also have the advantage that the operations invoked and the data items touched are known in advance.[3] We also call such transaction *static transactions*.

While on the theoretical level, the difference is slim, in practice the difference is important. First, this has an impact on performance [Mun98], but also certain problems (like deadlocks) can be avoided by using static transactions. Also dynamic transactions imply that the server executing the transaction has a communication channel with the client, so they can interact. For this reason, certain replication techniques can only handle static transactions.

2.4.5 Determinism

Another important issue for replication is determinism [Pol95]. Actual database systems use a wide range of non deterministic mechanisms, mostly for performance reasons. Database programs are multi-threaded, and the scheduling of threads is typically controlled by the blocking and unblocking of input/output operations, which are unpredictable. Also, databases rely heavily on time-outs to detect deadlocks. All this leads to a very non-deterministic system.

The problem is that determinism is an important assumption for replication. One common way of replicating data is to have all replicas process the same se-

[3] At least a superset is known in advance.

quence of instructions in the same order. This replication technique, called state-machine replication [Sch90], requires that the replicated object behaves in a completely deterministic way. Tools and techniques to make replicated applications deterministic are an ongoing research topic [NMMS99], but building a deterministic database system is a very important challenge and it is not clear if it is possible without losing too much performance. Issues related to determinism in databases when communication primitives are used have been studied in detail in [AAES97].

As requiring a fully deterministic database is not reasonable, other replication techniques that require no determinism have been proposed. Alas, those techniques require additional rounds of messages to make sure that the different copies of the data do not diverge, leading to bad performance. Some techniques have been designed to cover the middle ground: they do not require "full" determinism: execution must be deterministic to a point. To express the amount of determinism, we define the *point of determinism* [WPS+00a] as the point in the execution of a transaction after which the processing must be deterministic. Different replication techniques require different points of determinism.

2.4.5.A Point of Determinism

Formally, the point of determinism dp of transaction t, is the first operation of transaction t such as any operation o after dp executes in a deterministic way. This means that once dp is executed, the execution of the rest of transaction t is deterministic. The main implication is that once this point is reached the position of the transaction in the serial history can be determined.

The notion ofpoint of determinism is related to the notion of a serialisation point [BGMS92]. Serialisation points (sp) are the first operation in a transaction such as when the sp_1 of transaction t_1 executes before sp_2 of transaction t_2, then t_1 is before t_2 in the serial history. In other words, once the serialisation point of a given transaction t has been executed, then the position of transaction t in the serial history is known and fixed.[4] Therefore serialisation points are also points of determinism. The reverse is not true: a database might be deterministic, but reorder transactions in some way, thus offering no serialisation point. One reason for doing so would be to minimise the number of conflicts by using deterministic re-ordering [Ped99].

Note that the local database and its concurrency control determines the determinism point. A fully deterministic database would have its point of determinism at the first operation of the transaction. Strict two-phase locking is deterministic after the first phase: once all locks are acquired the system is deterministic, so the point of determinism is after the last read or write operation (if we consider dynamic transactions).

Databases that have an interface for an atomic commitment in order to support distributed transactions all have a point of determinism at the last operation of the

[4] We assume that the outcome (commit or abort) of the transaction is also known and fixed.

transaction. Once this operation is executed, the transaction is *ready to commit*, the system must be able to commit the transaction in a fixed position in the serial history, the transaction cannot be aborted.

This is a direct consequence of the relationship between serialisation point and points of determinism. The notion of serialisation points is used to determine if a database server can participate in a distributed transaction. If a database offers a serialisation point, then it can participate in a distributed transaction even without the explicit interface for an atomic commitment [BGMS92]. Database servers that offer an explicit interface for an atomic commitment protocol have a serialisation point. As all serialisation points are also points of determinism, techniques that can be controlled by an atomic commitment have a point of determinism.

If the database is completely non-deterministic, there is no point of determinism: as the dp is the first deterministic operation, this would be the operation after the termination operation (commit/abort). Some replication schemes typically require certain points of determinism. Because of this, some concurrency control schemes are unsuitable for certain replication techniques.

2.4.5.B Unilateral Aborts

One important aspect related to determinism is the issue of unilateral aborts. In normal operations, a database system can always decide to abort a transaction t as long as t has not committed. Reasons for aborting transactions are usually related to concurrency control, but can also be caused by issues like scarce resources (for instance not enough disk space), or a disk error. Replication techniques that require determinism cannot tolerate unilateral aborts and must therefore address such issues. For instance, they should have a very conservative resource handling policy (i.e. be sure to always have enough resources), but this is not always possible. Another option is to have the system use compensating transactions, that is, to run a new transaction that tries do redo the effects of the transaction that was unilaterally aborted. This technique is often used in federated databases to build distributed databases out of databases with no built-in support for atomic commitment. The problem is, compensating transactions must be carefully scheduled so that the resulting state is consistent.

Finally, one option is to artificially crash the server that did the unilateral abort. While this makes sense in some cases (for instance when a disk failure prevents the server from working), in other cases, an artificial crash is no warranted (for instance if a transaction aborts because of a temporary shortage of disk space). The overhead for crashing and then going through the recovery protocol is usually quite high. In general, introducing artificial crashes to solve control problems is considered expensive [CBDS01].

2.4.6 Optimistic and Weak Communication Protocols

The primitives introduced in Section 2.2.4 offer strong guarantees, and database replication schemes based on those primitives use those guarantees to enforce the ACID properties of the replicated database. Yet those strong guarantees come at some price in terms of performance. Protocols that implement weaker primitives have been proposed and specified – and promise better performance than their "stronger" counterparts. In certain cases, such "weaker" primitives are sufficient, and can be used in place of the "stronger" variants.

The properties of the communication primitives are usually used to ensure the isolation and atomicity properties of transactions (consistency and durability are usually enforced locally by each replica). But group communication primitives also have other benefits, like diminished deadlocks rates [HAA00]. If only one communication primitive is used, it must have strong properties to ensure isolation and atomicity. If two communication primitives are used (for instance total order broadcast and atomic commitment), then each primitive will only be responsible of ensuring a portion of those properties. So carefully chosen weaker communication primitives can be used. For instance some techniques rely on both total order broadcast and atomic commitment: isolation and atomicity are ensured by the atomic commitment, total order broadcast is used to decrease deadlocks. In this case, the total order broadcast can be weaker, as its properties are used mostly for performance.

Another way of using "weaker" protocols is to build optimistic protocols [Ped01]. Often, the weak protocol is a part of the strong protocol. For instance a uniform total order broadcast can be seen as a non-uniform total order broadcast with an additional communication round. So a first guess of the outcome of a total order broadcast can be delivered early, and confirmed later. In between processing can take place [KPAS99].

The most promising weak protocols are non-uniform total order broadcasts and best-effort total broadcast.

2.4.6.A Non-Uniform Total Order Broadcast

Non-uniform total order broadcast [WS95] is defined by weakening the total order and the agreement properties:

Non-Uniform Agreement If a *green* process R-delivers a message m, then all *green* processes eventually R-delivers m.

Non-Uniform Total Order If two **green** processes p and q A-deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'.

The difference between uniform and non-uniform total order broadcast is that in the non-uniform case, a process that is not green might deliver messages in the

wrong order, or deliver messages that will not be delivered by all others. Typically this means the process can deliver messages in the wrong order and then crash.

2.4.6.B Best Effort Total Order Broadcast

Best effort total order broadcast offers even weaker guarantees than non-uniform total order broadcast. Basically a best effort total order broadcast will "try" to respect the specifications, but might not enforce them in certain circumstances. Best effort algorithms are easier to implement, because they can be based on model assumptions that are true only most of the time.

For instance the timed asynchronous model [CF99] assumes that clocks are available and that bounds on messages transmission and processing times are known, like in the synchronous model; however those bounds do sometimes not hold, which is a called a performance failure. In this context, any algorithm based on timing assumptions can be used [CASD85]. As long as the timing assumptions hold, total order will be ensured. As periods of instability where performance failures occur are rare, the total order broadcast primitive will work correctly most of the time – which is good enough if total order broadcast is used to increase performance.

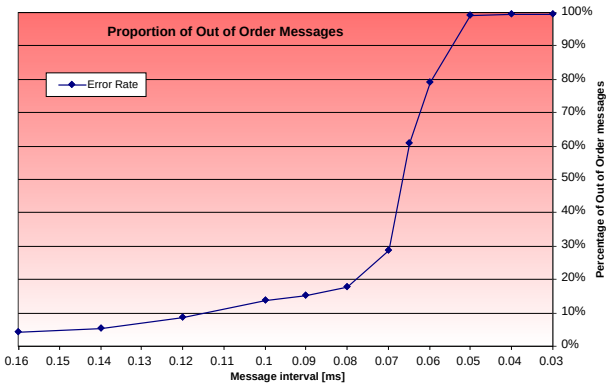

Figure 2.7: Spontaneous ordering in a LAN

Another example of best-effort algorithm for total order broadcast is based on the observation that messages in a LAN are delivered in total order most of the time. This is due to the fact that in an Ethernet backbone, only one message can be transmitted at the time, so the backbone enforces a total order.

Figure 2.7 shows the observed proportion of spontaneously ordered messages in LAN setting. The X axis represents the interval between two messages sent, and the Y axis represents the percentage of messages that were delivered out of order. This experiment was done on a cluster of 11 Pentium III/766 MHz machines with 128 MB or RAM and a 100-Base TV full duplex Ethernet network interface interconnected with an Ethernet hub. The experiment was run using the Neko [UDS01]

framework. We see that if the load is less than one message every 0.15 *ms* the spontaneous ordering is above 90%. Another interesting result of this experiment was that if messages are delivered out of order, the "error" in the sequences is always one or two positions. I.e, if the sequence of messages is $m_i, m_{i+1}, m_{i+2}, m_{i+3}$ then, in the worst case, the spontaneous delivery order is $m_{i+1}, m_{i+2}, m_i, m_{i+3}$. We call the amount of reordering the *message reordering level*.

In those conditions a best effort implementation of total order broadcast simply relies on the underlying total order, which is correct "most of the time". The spontaneous ordering property can also be used to optimise certain replication techniques, this is discussed in Section 5.3.6.

2.5 Summary and Synthetic Model

2.5.1 Summary of Both Communities

	Database Systems	Distributed Systems
Motivation	Performance	Fault-Tolerance
What is Replicated	Data	Processes
Operation Type	Multiple (Transactions)	Single (Remote Invocations)
System Model	Synchronous	Synchronous Asynchronous etc...
Consistency	1-copy Serialisability	Linearisability

Table 2.3: Summary of both communities

Table 2.3 summarises the differences between both communities. Each line describes one aspect of replication and how both communities approach it. The motivation for replication is different: in one case replication is a mean to get fault-tolerance, in the other it is a way to achieve better performance. The second difference is what semantic element is replicated: in a distributed system context, the unit of replication is a process, that is, a functional entity that does some calculation. In the database context, it is data that is replicated. The operation type considered is also different, distributed system models consider single events (like remote procedure invocations (RPC)), while database systems consider groups of operations (transactions). The network model is also different: database systems are usually built on a synchronous model, while distributed systems rely on multiple different models (synchronous, asynchronous, failure detectors etc...). Finally, the consistency criteria are also very different. In the distributed system community, application level consistency is not often considered, so consistency is defined

regarding the delivery of events, typically causal order, view synchronous order or total order. Linearisability is an application level consistency criterion sometimes used in distributed system [HW90]. Linearisability is stronger than serialisability.

2.5.2 Synthetic Model

We have presented the models for the database module (Section 2.3) and the communication module (Section 2.2). We now need to define the model for the overall system: in our case, the whole replicated database system. The goal here is not to replace the models described the previous sections, but to integrate them in a larger model that takes into account the practical aspects described in Section 2.4. Figure 2.8 illustrates the synthetic model.

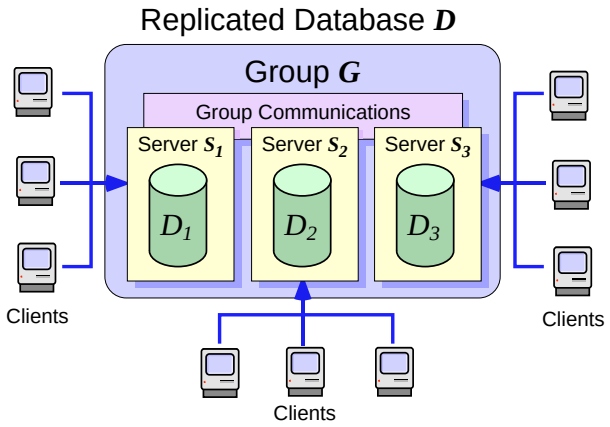

Figure 2.8: Synthetic model

We consider a database D distributed on servers $\{s_1 \dots s_n\}$, where each server s_i contains a full copy of database noted D_i. Each server s_i contains a database server capable of executing transactions on the local copy of D. Each database server is a full database system, capable of ensuring the ACID properties on the transactions it executes.

We consider that all database servers $\{s_1 \dots s_n\}$ form a group G, and have access to all group communication primitives described in Section 2.2.1. So each server is also a process for the group communication system. We call the combination of server and process a *replica*. Depending on the model, a replica might have one attached process (crash-recovery model) or multiple incarnations of the same process (crash no-recovery model). Figure 2.9 shows the relationship between the database server and the group communication process for one server machine. The left part shows a crash scenario in the dynamic crash-recovery model. Both the database server and the group communication process crash when the machine crashes. Upon recovery, both the database server and the group communication process recover. The right part shows the same scenario in the dynamic crash no-

recovery model. In this case, upon recovery, the group communication process does not recover, instead, a new incarnation of this process is created and joins the group. This distinction has some implication on the recovery process and is discussed in Chapter 4.

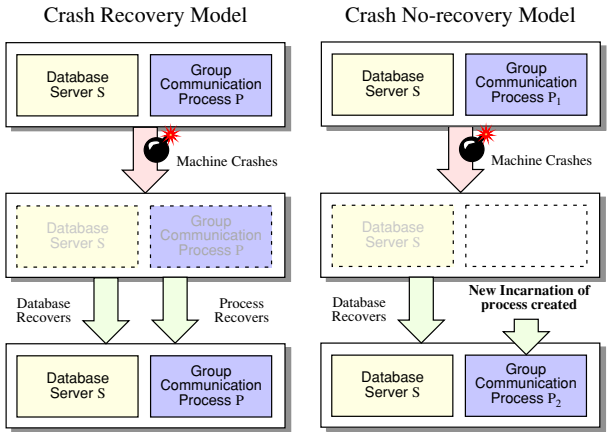

Figure 2.9: Relationship between database server and group communication process

We also consider clients. Clients are not part of the group G and do not rely on group communications. Clients only have access to point-to-point communications. Clients interact with the database D by sending transactions. Those transactions can be either stored procedures or interactive transactions (Section 2.4.4). To send a transaction, client c connects to server s_c. The client c selects s_c by choosing it among a list of servers. This list is static, and not updated by the group communication system. It contains all potential servers in the system. If a client does not get a response from some server, it will try another. If a server cannot handle a client's request (for instance because it is recovering, or because it is acting as a back-up), it will refuse to handle the request.

When client c submits transaction t_c to s_c, we call s_c the delegate for transaction t_c. The delegate is responsible for processing t_c on the database D on behalf of client c. If the transaction is interactive, all interactions will be done between the client and the delegate. If the transaction is a stored procedure, then the delegate is responsible for sending the result of the transaction back to the client. The client is not aware of the replication and only interacts with the delegate server.

The replicated database appears as one single non-replicated database to the client – it guarantees *one-copy serialisability*[5] and the ACID properties. The system hides any failure that is not the failure of the delegate or the client. If the client crashes before the end of the transaction, the delegate aborts the transaction. If the delegate crashes, the transaction can either commit or abort: the client needs to

[5]We do not consider quorum systems here.

connect to another delegate in order to check the outcome of its transaction.

Chapter 3

Classification of Replication Techniques

Crude classifications and false generalisations are the curse of organised life.

George Bernard Shaw

Numerous database replication techniques, from both the database community and the distributed system community have been proposed in the literature. Yet comparing the protocols developed in the two communities is a frustrating exercise. Due to the many subtleties involved, mechanisms that are conceptually identical, end up being very different in practice. So, it is very difficult to take results from one area and apply them in the other or to relate replication techniques. Another issue is that while many possibilities have been explored, it is difficult to see if all possibilities have been considered.

One good way to understand and compare replication techniques is by classifying them. This makes it possible to see what techniques are related and to verify that the solution space was completely explored.

Probably the best known classification of database replication techniques has been proposed by Gray *et al.* [GHOS96]. This classification uses two criteria: where a transaction can be initiated, and in what context it will be executed. The classification is summarised in Figure 3.1. The first criterion, *object ownership* describes what nodes "own" the data. Only a node that owns the data can accept updates to it. This criterion decides to which node of the system transactions can be submitted. If the data is owned by all nodes in the system, then transactions can be submitted to any node. If the data is owned by a master, then transactions need to be submitted to this master. The group ownership category is also called *update everywhere* replication (or sometimes *peer-to-peer*) while the the master ownership category is also called *primary-copy* or *primary/backup*.

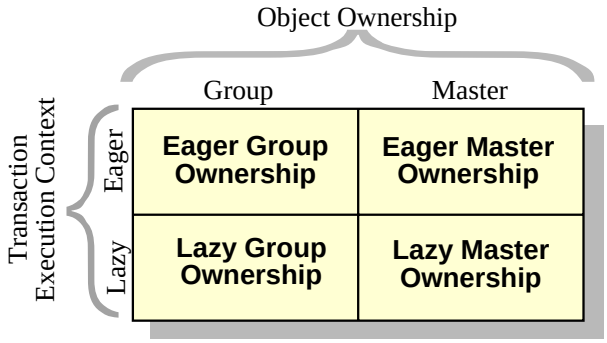

Figure 3.1: Gray's classification

The second criterion defines in what scope transactions are executed. In the case of *eager* replication, the updates of all replicas are done in the scope of one distributed transaction: the transaction can be aborted at any point, by any copy. This guarantees that all replicas stay consistent – this is also sometimes called *synchronous* replication. In the case of *lazy* replication, the transaction updates one replica, and the changes are shipped to other replicas outside of the scope of the transactions. This can lead to an inconsistent system in case of failure, but can also violate the ACID properties even without failures [Gol95]. This replication scheme is also sometimes called *asynchronous* replication.

While useful, this classification is rather coarse and also has trouble coping with *non-voting* replication techniques. Those techniques do not use atomic commitment, and have no final synchronisation phase at the end of the transactions, so they should be classified as *lazy*. On the other hand, they ensure that the execution of a transaction is serialisable (and should therefore be allowed to commit) before actually committing it, so they are also *eager*. This has led some people to call them consistent lazy techniques [HAA99a].

Other existing taxonomies of data replication techniques take into account a broad spectrum of replication schemes, including many with weak consistency and availability properties, but either without including techniques based on group communication [CHKS94], or considering only simple cases [CP92]. In this chapter we describe an abstract classification that focuses on eager replication techniques and emphasises the synergy between communication and transaction management. By *eager* replication, we mean techniques that enforce serialisability before committing a transaction (so non-voting techniques are included). There are two reasons for focusing on eager replication. First, eager replication is where the use of group communications makes sense: group communication offer strong guarantees, so they are useful to build a replicated system with strong guarantees. Second, it is difficult to compare strategies that relax consistency. While models with relaxed serialisability have been proposed [KLW96, KB94], the tradeoff between performance and consistency is not clear: how much consistency gives you

how much performance? How do you measure the loss of consistency? Models with relaxed serialisability have been proposed but their relationship with lazy replication is not yet very clear.

The benefits of our classification effort are numerous. First, it allows us to identify the key components of a database replication protocol. Second, it helps us to better understand the role played by each component and its influence on the nature of the protocol. Some of the strategies we consider have been described in the literature, but not all. Third, the classification forms the basis for quantitative comparison of the various replication strategies identified. This quantitative comparison will be presented in Chapter 5. These quantitative comparisons will shed light on many different aspects of eager replication and the role that transaction management and group communication play in implementing eager replication protocols.

3.1 Classification Criteria

Eager replication protocols can be organised according to three parameters that determine the nature and properties of the protocol and, in some cases, also its performance. These parameters are: the server architecture (Section 3.1.1), how changes or operations are propagated across servers (Section 3.1.2), and the transaction termination protocol (Section 3.1.3).

3.1.1 Server Architecture

This parameter matches Gray's classification. We distinguish between update everywhere (group ownership) and primary-copy (master ownership) replication techniques.

3.1.1.A Primary-Copy

This replication technique requires to have a specific site – the *primary-copy* – associated with each data item. All other sites are considered as *backups*, hence the name primary/backup.

Any update to the data item must be first sent to the primary-copy where it is processed (executed or at least analysed to establish its serialisation order). The primary copy than propagates the update (or its results, see Section 2.4.3) to all other sites.

Primary-copy techniques are found in both distributed system community and database community [WPS+00b]. In the distributed system community, it is called *passive replication* [BMST93, GS97] and relies on group communications mechanism (more specifically the view) to select the primary and to ensure that only one primary is present in the system. In the database community those issues are addressed in an ad hoc way.

As expected, primary-copy approaches introduce a single point of failure and a bottleneck. These limitations can be solved by making the protocol more complicated. Thus, if the primary crashes, one of the other servers takes over the role of primary, which requires an election protocol. To avoid bottlenecks, databases do not make a single site the primary for all data items. Instead, the data is partitioned and different sites become the primary for different data subsets. In what follows, we will mostly ignore these aspects of the protocols since they are orthogonal to the discussion.

3.1.1.B Update Everywhere

Update everywhere replication allows updates to a data item to be performed anywhere in the system. That is, updates can concurrently arrive at two different copies of the same data item (which cannot happen with primary-copy). Because of this property, update everywhere approaches are more graceful when dealing with failures since no election protocol is necessary to continue processing. Similarly, in principle, update everywhere introduces no performance bottlenecks. However, update everywhere may require that instead of one site doing the work (the primary-copy) all sites do the same work – so in some sense, the bottleneck is not eliminated, but replicated. If one is not careful with the design, update everywhere may affect performance much more than primary-copy approaches.

3.1.2 Server Interaction

The second parameter to consider is related to the degree of communication among database servers during the execution of a transaction. This determines the amount of network traffic generated by the replication algorithm and the overall overhead of processing transactions. This parameter is expressed as a function of the number of messages necessary to handle the operations of a transaction (but not its termination). The number of network interactions will impact the design of the replication protocol. This design will affect what properties the different communication protocol will have to offer (order, uniformity etc.). We consider two cases: constant interaction and linear interactions. So if m_t is the number of operations of a transaction t and k_t the number of network interactions, we consider the cases where $k_t = \mathcal{O}(1)$ and $k_t = \mathcal{O}(m_t)$.

3.1.2.A Constant Interaction

Constant interaction corresponds to techniques where a constant number of messages is used to synchronise the servers for a given transaction, independently of the number of operations in the transaction. Formally, this means that $k_t = \mathcal{O}(1)$. This single interaction might be a simple broadcast message, or an advanced protocol with complex properties, like total order broadcast. Replication techniques in this category might also use a constant number of messages instead of one (typically one high-level message can imply many low-level messages). In the rest

of this discussion we will consider that constant interaction replication techniques have one single interaction. This simplifies discussion without reducing the model.

Replication techniques in this category exchange a single message per transaction by grouping all operations of the transaction in this message. When this interaction takes place depends on the nature of the transaction. If the transaction is interactive (see Section 2.4.4), the client and the delegate communicate during the execution of the transaction, therefore the transaction can only be known and bundled in a message when the clients asks the delegate to commit the transaction. If the transactions are static, then this exchange can take place anytime.

3.1.2.B Linear Interaction

We describe a technique as having a linear number of interactions, if $k_t = \mathcal{O}(m_t)$, that is, if the number of network interactions k_t to handle a transaction t is proportional to the number of operations m_t in transaction t. Replication techniques in this category typically propagate each operation of a transaction on a per operation basis. The operations can be sent either as SQL statements (high-level transactions) or as log records containing the results of having executed the operation in a particular server (low-level transactions). Here again, to simplify the discussion we consider that a linear interaction replication technique uses m_t messages, where m_t is the number of operations in transaction t.

3.1.3 Transaction Termination

The last parameter to consider is the way transactions terminate, that is, how atomicity is guaranteed. We distinguish two cases: voting termination and non-voting termination. This parameter distinguishes replication techniques that need some message exchange to ensure the ACID properties, and techniques where those properties can be ensured without explicit termination messages.

This factor is independent of the number of interactions. Yet one might argue that transaction termination is simply one additional network interaction between all copies of the database and should be counted as such. This factor would not affect the order of the number of network interactions (constant or linear), and has no influence and should therefore be discarded.

While this reasoning is legitimate, this last interaction has a important impact on the design and the performance of database replication schemes and should therefore not be simplified away. Both lazy replication schemes and group communication-based schemes promise better performance by removing or simplifying this phase. The termination phase typically implies synchronisation between the replicas, but also operations on stable storage. Both operations are expensive in terms of performance. The presence or absence of the termination phase, and its complexity is therefore an important aspect of database replication.

Transaction termination is traditionally associated with two properties of replication: hot/cold standby and 1-safe/2-safe. If a primary-copy replication scheme

has no voting termination phase, there will be little or no flow control and the backup will tend to be cold standby backups (see Section 2.4.3), while a voting phase will help the backups to be in a hot-standby mode.

The transaction termination criterion is also related to an established way of classifying primary copy database replications [GR93, PGM94]. In this context, system are distinguished by the number of sites where a transaction is guaranteed to have committed when the client receives commit confirmation. In *1-safe* techniques, the transaction is guaranteed to have committed on one site (the primary/delegate). In *2-safe* techniques, it is guaranteed to commit on all sites (the primary and the backups). Traditionally, techniques are considered *2-safe* if the replication protocol uses voting. We will show in Chapter 4 that the issue is more complex.

3.1.3.A Voting termination

Voting termination [ÖV99] requires an extra round of messages to coordinate the different replicas. This round can be as complex as an *atomic commitment protocol* like those described in Section 2.2.4.E or as simple as a single confirmation message sent by a given site (usually the delegate/primary) to the other sites.

The difference between an atomic commitment protocol and a simple message has an impact on the database replication scheme. A simple message means servers that are not the delegate might not do unilateral aborts. So the termination phase is not simply present or absent. A replication scheme might have a *strong* voting termination phase (atomic commitment), or a *weak* one (simple message). Strong voting implies that decision to abort a transaction is *multilateral*; each replica might cause the abort of a transaction. Weak voting means that this decision is *unilateral*: only the primary (delegate) can decide to abort the transaction.

Another important aspect of a voting termination phase is *flow-control*. Without a strong voting phase, servers cannot tell a delegate they cannot handle the load. Because of this, one slow server might not be able to cope with the load and end up with more and more pending transactions and no means to request that other servers slow down. This situation might eventually lead one server to become overloaded and fail. The atomic commitment protocol forces all servers to wait, so the system will naturally synchronise. The main drawback of voting, especially strong voting, is that the system always waits for the slowest replica.

3.1.3.B Non-voting termination

Non-voting termination implies that sites can decide on their own whether to commit or abort a transaction. Non-voting techniques require replicas to behave deterministically – so replicas need to enforce atomicity (all replicas abort a transaction, or all commit it) and one-copy serialisability (all replicas execute transactions in the same serial order). This, however, is not as restrictive as it may appear at first glance, since the determinism only affects transactions that are serialised with re-

spect to each other. Transactions or operations that do not conflict can be executed in different orders at different sites. Determinism is discussed in Section 2.4.5.

3.2 Replication Techniques

In this section, we explore all the combinations that result from the classification parameters in Section 3.1. All technique categories are described in five parts: description, references, requirements, discussion and a summary. The *description* paragraph describes the general framework of the replication techniques. Existing replication techniques that fit in this category are listed in the *reference* paragraph, along with the relevant bibliographic references. The *requirement* paragraph describes the requirements needed to build a replication technique fulfilling the classification criteria. Those requirements are expressed either for the communication infrastructure, or for the database system on each server. Requirements on the communication system are usually ordering or uniformity constraints on the delivery of messages to database servers. For the database system the requirement is determinism which is expressed in terms of the point of determinism.

3.2.1 Update Everywhere

In update everywhere techniques, the clients can send their requests to any server. The server contacted will act as the *delegate* for the requests submitted by the client. The delegate will process the requests and synchronise with the other servers to ensure one copy serialisability.

3.2.1.A Update Everywhere – Constant Interaction – Non-Voting Techniques

Description. Figure 3.2 shows the basic structure of such a replication technique. In the discussion that follows, we assume that there is only one network interaction between servers. This simplification makes the description clearer and does not leave out any important detail. The protocols in this category execute according to the following steps:

1. The transaction starts on the delegate server.
2. The transaction is processed in a non-deterministic way.
3. The *point of determinism* is reached.
4. The transaction is sent to all servers using an atomic commitment.
5. Processing continues on all replicas in a deterministic way.
6. Each replica terminates the transaction in the same way.

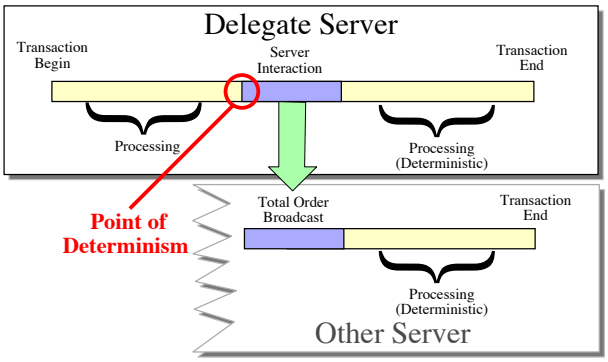

Figure 3.2: Update everywhere, constant interaction, non-voting

References. Previous works following the Update Everywhere – Constant Interaction – Non-Voting model can be divided according to where the point of determinism is placed.

If the point of determinism is at the beginning of the transaction, the whole transaction processing is deterministic, and the role of the delegate server is simply to forward the transaction using the total order broadcast primitive (step 2 in the description above does not really apply). The delegate simply acts as a *proxy* for the client, contacting all servers to process the client's request. One possible optimisation would be that the delegate transforms the high-level transaction into a low-level one, so that this work is only done once. Overall, this approach is close to *active replication* [Sch90]. An early example of such an approach can be found in [PGM89]. Other techniques in this category are presented in [Kei94]. The technique described in [KPAS99] also uses total order broadcast and an early point of determinism with certain forms of optimistic transaction execution.

Techniques with the point of determinism at the end of the transaction processing are called certification techniques. In this case, the whole transaction is handled in a non-deterministic way by the delegate, and only the last stage of the processing, the *certification*, is deterministic. This certification stage is executed at all sites, and allows to decide if a transaction will be committed or aborted. In [PGS98] information about both read and write operations is sent to all sites in order to detect conflicts during the certification phase. The protocol A4 in [HAA99b] uses a similar approach. Techniques in [KA98, KA00b] use snapshot isolation instead of serialisability to avoid conflicts between read and write operations, and hence, the certification phase is restricted to write operations. In all cases, the certification phase is deterministic.

Requirements. We discuss correctness by distinguishing transaction atomicity (i.e., all or none of the copies of the database commit a transaction) from the other correctness criteria.

Independently of where the point of determinism lies, the mechanism used to guarantee one copy serialisability is always the same. The total order used in the broadcast acts as a guideline to all sites. Each site guarantees that its local serialisation order will follow the total order, and thus all sites will produce the same serialisation order (since they see the same total order). The differences in the protocols lie on the determinism point. For protocols that place the determinism point at the beginning of the transaction, the total order suffices. For protocols that place the determinism point at the end of the transaction, things are a bit more complicated. In particular, when confronted with situations where a transaction needs to be aborted, a delegate server can only abort transactions not yet seen by other sites. The protocol must ensure that as soon as transactions are seen by other sites, there will be no problems with their scheduling or that all sites will end up aborting the transaction.

Related to this, techniques in this category do not need a distributed deadlock detection system. Since transactions are sent in one step using a total order broadcast, locks for the whole transaction can be acquired atomically and in the same order at all sites thereby preventing distributed deadlocks.

Transaction atomicity is enforced by a uniform reliable broadcast of the messages, and the deterministic behaviour of the different servers. This guarantees that whenever a server delivers a message and commits the transaction, each server will deliver the message (uniformity) and commit the transaction (determinism).

Discussion. The use of uniform reliable broadcast for transactions delivery has the advantage that a site can commit a transaction as soon as it is delivered without waiting for other sites. This can result in improvements to the response time since sites are not tightly synchronised – because locks can be released as soon as the transaction finishes; lock contention is also lower.

A point of determinism at the beginning of the transactions usually implies that the datasets of the transactions are known in advance (static transaction). Thus, what it is being sent to all sites is typically an invocation of a stored procedure. It follows that all sites must execute this procedure deterministically since there is no voting phase. This is one of the limitations of these approaches, since making all sites completely deterministic can be difficult in practice.

Using points of determinism at the end of the transactions imposes a lower burden on the database system. A late point of determinism does not require static transactions: writes can be differed to the end of the transaction or executed on a shadow copy, Nevertheless, the fact that the point of determinism comes at the end of the transaction has several implications. The main one is that there is a degree of optimism in the execution. Servers accept many transactions but might abort some of them. There is a tradeoff between early points of determinism and abort rate. Having the point of determinism early means low aborts due to conflicts. Having this point late implies having higher chances of conflicts. Intermediate solutions, where the point of determinism is in the middle of the transaction would

be a compromise. One way of implementing such a solution would be to execute all the reads at the beginning of the transaction, and then do all the writes in an atomic and deterministic step. In this case, the point of determinism would be the first write operation.

Summary. Replication techniques based on update everywhere with a constant number of interactions and non-voting require total order broadcast. From the database point of view, the necessary determinism is achieved by ensuring that the local serialisation order matches the total order and by resolving conflicts in a deterministic way once transactions go beyond their point of determinism.

3.2.1.B Update Everywhere – Constant Interaction – Voting Techniques

Description. Figure 3.3 shows the basic structure of a replication technique in this category. This technique is similar to the one in the previous section, in that all the interactions are done using one communication phase. Additionally, a final voting phase is executed at the end of the transaction's execution to ensure that all replicas agree on the outcome. The execution is done in the following way:

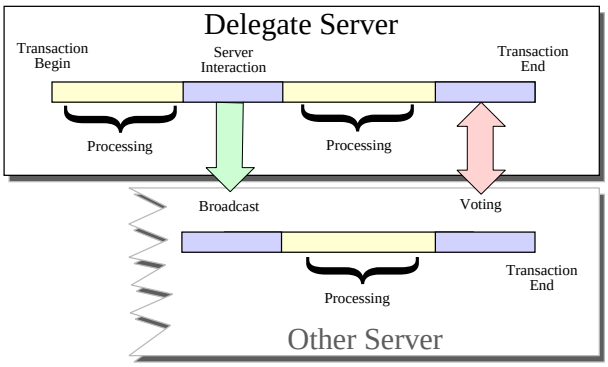

Figure 3.3: Update everywhere replication, constant interaction, voting

1. The transaction starts on the delegate server.
2. The transaction is processed in a non-deterministic way.
3. The transaction is broadcast to all servers.
4. Processing continues on all replicas.
5. A voting termination phase takes place.
6. Each replica terminates the transaction according to the outcome of the voting protocol.

References. As an example of this technique, in [FC94] the delegate server broadcasts a transaction to the other sites immediately when it is submitted and a total order broadcast is used (the total order being implemented using synchronised clocks). Also here, the total order is used as a guideline at every site to serialise transactions. The final voting phase is only used to ensure atomicity in the case of different types of failures. Because these failures can occur at any site, an atomic commitment is needed.

Another example of constant interaction with voting is the protocol called "serialisability based protocol", presented in [KA98, KA00b] and protocol A3 presented in [HAA99b]. In those protocols, the transaction is locally executed at the delegate site and then sent to the other sites using a total order broadcast primitive. Here, the delegate site is the only one to decide whether the transaction can commit or must abort. Because the situation leading to an abort (due to serialisation problems of local reads and global writes) is not seen by all sites, the delegate site needs to communicate the decision to all other sites. This means that the voting is weak: a single message that indicates whether the delegate server has committed or aborted the transaction. As a consequence, while the delegate site has the choice to commit or abort the transaction unilaterally, the other sites must behave deterministically in the sense that they have to obey the commit/abort decision of the delegate.

A third example is the optimised form of 2PC described in [BHG87]. In this protocol, write operations are deferred to the end of the transaction, and the first phase of 2PC (vote request) also contains the transaction updates. Participants in the protocol respond with a *yes* vote if they can obtain the locks for those updates. Otherwise they respond *no* and the transaction is aborted. In this case, the only interaction is the enhanced version of the 2PC protocol.

Finally, techniques in this category can be related to semi-active replication [PCD91]. In this replication technique, all process deliver the same requests, but if some non-determinism appears, a *leader* is chosen and this leaders sends the result of its execution to all other processes (called *followers*). If such a system would handle transactions, this would mean that in case of doubt, the leader does not send to the followers the outcome of the transaction (commit or abort), but the serialisation order that was chosen. Of course the leader could also decide to abort the transaction and send the result (abort) to the *followers*.

Requirements. In principle, protocols of this type could use any form of broadcast primitive (including unreliable broadcasts). However, the type of broadcast primitive used determines the voting phase. If the primitive cannot guarantee that all sites will do the same processing, then the voting phase can only be an atomic commitment and, as part of this phase, discrepancies among sites must be resolved. Furthermore, distributed deadlocks might occur and must be resolved. If the broadcast is totally ordered, then the requirements are similar to those of Section 3.2.1.A (e.g., also no need for global deadlock detection).

How atomicity is guaranteed depends on the protocol. If atomic commitment is used (strong voting) then it guarantees the atomicity. If the technique relies on weak voting, the primitive used for broadcasting the transaction must be uniform – so if a single broadcast ensures atomicity, then this message must be sent using a uniform reliable broadcast.

Discussion. Having a voting phase relaxes the determinism requirements on the database servers. In practice, and given that complete determinism in a database server is difficult to achieve, many protocols include a voting phase in one form or another. Nevertheless, some limited form of determinism and the use of a total order broadcast considerably simplifies the protocols.

Checking if the all the copies have executed the transaction in the same serial order at commit time implies a large waste of resources if the serial orders do not match. Conflicting serialisation orders on the different replicas are the cause of the rising deadlock rate that was described as the main problem in eager replication [GHOS96].

While a total order broadcast will still help to decrease conflicts, having a voting phase consisting of a complete atomic commitment allows to relax the requirements on the total order. If the total order is not strictly enforced, the atomic commitment phase will detect discrepancies. Therefore if one of the properties of the total order is not enforced, then this will be detected and corrected by the atomic commitment. This means that a weak total order broadcast (either non-uniform or best effort) can be used.

Summary. The requirements are intertwined. If the broadcast does not impose a total order, then the voting phase must be an atomic commitment and will involve some form of certification. If total order broadcast is used, then the voting phase can be significantly simplified (one confirmation message) as long as sites are deterministic and can obey the unilateral decision of the delegate server.

3.2.1.C Update everywhere – Linear Interaction – Non-Voting Techniques

Description. This category is somewhat misleading. Non-voting implies that there is no round where the fate of the transactions can be agreed upon. Therefore, these protocols must be fully deterministic. Unlike for constant interactions, there is no possibility to send the transaction as a whole. Sending operations one at a time requires that all sites treat them in exactly the same way. Nevertheless, at the end the delegate site has to indicate that the transaction has finished. This implies that there is a *termination* message. Assuming this termination message is not used for voting, the general structure of techniques in this category are outlined in Figure 3.4:

1. The transaction starts on the delegate server.

2. The first operation is sent to all servers using a total order broadcast.
3. The first operation is executed on all servers.
4. Items (2) and (3) are repeated until the transaction ends.
5. The delegate sends a termination message to indicate the end of the transaction.

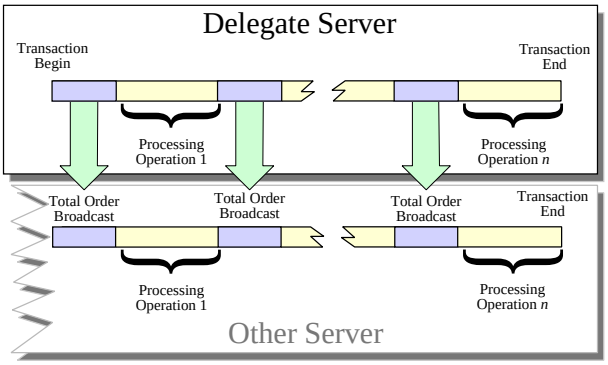

Figure 3.4: Update everywhere, linear interaction, non-voting

References. An example of a technique of this category is presented in [AAES97]. Each operation (reads included) is broadcast (total order) to all sites, and sites must behave deterministically in order to react identically to each operation. Techniques A1 and A2 presented in [HAA99b] also fit in this category. Techniques in this category are very similar to replicated persistent objects [LS98]. Replicated persistent objects are modified by invocations. Those invocations are delivered in total order to all objects and processed deterministically. Multiple invocations can be part of a transaction. The transaction terminates when the last invocation of the transaction is processed.

Requirements. Since there is no voting phase, atomicity can only be guaranteed by sending operations using total order broadcast broadcast. one-copy serialisability is the result of the local concurrency control mechanism used at each site, and the determinism across sites. The total order broadcast needs to be uniform only when sending the last operation. The problem with non uniform total order broadcast is the risk of delivering a message out of order and then crashing. Consider transaction $t = \{op_1, op_2, \ldots op_i\}$. If one server delivers op_i out of order and then crashes, this might lead to an incorrectly serialised transaction to be committed, thus violating one-copy serialisability. So the last total order broadcast must be uniform. If one server delivers op_{i-1} out of order and then crashes, t will never

be committed with the wrong serial order, instead it will first be aborted (with the wrong order) and then restarted (with the correct order). Therefore all total order broadcasts except the last can be non-uniform.

Because sites are fully deterministic, there are no distributed deadlocks: all sites have the same wait-for graph, so cycles in this graph cannot imply more than one site. Local deadlocks must be assumed to be resolvable in a deterministic fashion [AAES97].

Discussion. This technique has the major drawback of requiring absolute determinism on all sites, which is a very strong requirement. In addition, there is considerable network overhead since each operation results in a totally ordered broadcast. In general, this technique has not been pursued in the literature as a viable option.

Summary This technique requires each operation to be broadcast using a reliable total order broadcast. The last operation must be broadcast using a uniform total order broadcast. Moreover, transaction processing must be fully deterministic.

3.2.1.D Update everywhere – Linear Interaction – Voting Techniques

Description. This form of database replication technique is the most studied in the literature. Among its many variations, one of the best known is the read-one-write-all technique [BHG87]. Figure 3.5 shows the interactions of techniques in this category:

1. The transaction starts on the delegate server.
2. Each operation is broadcast to a quorum of sites.
3. Each operation is executed on its quorum.
4. Items (2) and (3) are repeated until the transaction ends.
5. A voting protocol is executed.
6. Each replica terminates the transaction according to the outcome of the voting protocol.

References. This category includes all the traditional database replication protocols: read-one/write-all (ROWA), write-all-available, and quorums [BHG87]. Most of the effort in this area has been devoted to provide different ways to build quorums. Good surveys of early solutions are [DGMS85, BHG87]. Later approaches mainly optimise quorum sizes and communication costs or analyse the trade-off between quorum sizes and fault-tolerance [KS93, RST95, TP98]. In [SAA98] multicast primitives with different ordering semantics are used. The

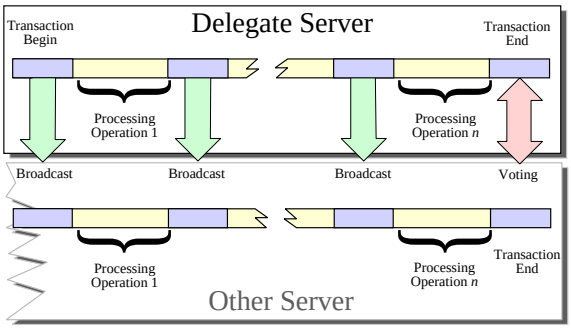

Figure 3.5: Update everywhere, linear interaction, voting

authors propose algorithms using reliable broadcast or causal broadcast which require an atomic commitment to guarantee serialisability. In technique A3 described in [HAA99b], the delegate uses a total order broadcast to send all write operations to the other replicas and a final broadcast to send the decision to commit or abort hence implementing a weak voting protocol. In [RJP01] an optimistic version of atomic commitment is used to reduce latency.

Requirements. One-copy serialisability is achieved by executing each read operation on a read quorum of replicas, each write operation on a write quorum. With this, each site follows a local concurrency control protocol that guarantees serialisability, typically 2-phase-locking [Tho79] or time-stamp based algorithms [BSR80]. Atomicity is guaranteed by an atomic commitment (typically 2PC) during the voting phase.

If total order broadcast is used to send operations to all replicas, then the technique is conceptually similar to the technique described in Section 3.2.1.B, that sends all writes in one message and uses a weak voting phase. The only conceptual difference is that the sending of operations is spread on n totally ordered messages instead of one. The requirements for the weak voting messages stay the same. In this case, operations are send using a non-uniform total order broadcast, and the last voting message is sent using reliable uniform broadcast.

Discussion. This technique is very well understood. However, in spite of the amount of work invested in this technique, it is not very relevant in practice [GHOS96]. The reason is that it has a high overhead (because of the linear number of messages) and has proven to significantly limit scalability due to deadlocks. In addition, when the voting phase involves a complete atomic commitment, the client only gets the response once all replica are ready to commit the changes: this can result in very long response times.

3.2.2 Primary-Copy

In primary-copy techniques, the clients must send their requests to one particular server. What this means is that there is only one server acting as a delegate for all clients at any point of time; this server is called the *primary*. Because there is only one server executing the transactions, conflict between transactions are detected and solved in one place. So there are no global transaction conflicts nor distributed deadlocks. The only thing that has to be assured is that there is only one primary in the system at any time. The primary-copy approach is widely used in eager replicated databases to minimise conflicts among transactions executed over replicated data.

Only the group of servers needs to know what server is the primary. Clients have no access to this information (see Section 2.4.1), and simply try all the servers in their list. If a server is requested to handle a transaction and is not the primary, it will reject the request and send back the identity of the current primary. Having all the servers agree on the current primary is a problem that can be solved using consensus. If the group communication system offers a *view* then the primary can be chosen by taking the first member of the view. A similar mechanism can be built using perfect failure detectors.

From now on we will refer to the sites that are not the primary copy for a data item d as *backups* of d. Primary copy techniques are often also called primary/backup techniques. The backups only install the changes sent by the primary. There is a distinction between *active* and *passive* backups. An active backup is the primary for a subset of the data, and the backup for the other data. A site that is not the primary for any data is a *passive* backup. The main issue with active backups are transactions that update data of multiple primaries. Replication techniques that support such transactions have the same requirements that update-everywhere replication: the execution of a transaction must be synchronised on all replicas. For this reason we will concentrate of *passive* backup replication schemes in this section.

3.2.2.A Primary-Copy – Constant Interaction – Non-Voting Techniques

Description. Techniques in this category are generally used for cold-standby replication. The protocols have the following general outline (see Figure 3.6):

1. The transaction is executed at the primary.
2. When the transaction terminates, the corresponding log records are sent to all backups using a FIFO reliable broadcast.
3. The primary commits the transaction without waiting for the backups to install the changes.
4. The backups eventually install the changes.

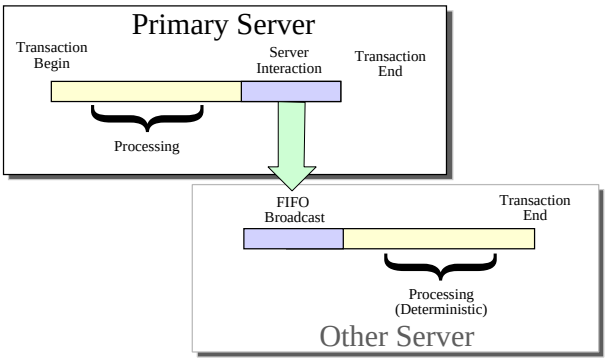

Figure 3.6: Primary-copy, constant interaction, non-voting

The concrete nature of the protocol depends on the type of broadcast primitive used. In its simplest form, the protocol is based on FIFO delivery, in order to ensure that transaction changes are installed at the backup in the same order they were executed at the primary.

References. Techniques in this category are typically described in the database literature as 1-safe, cold standby, primary backup replication techniques [Lyo88, Lyo90, GR93, PGM94]. Passive Replication [BMST93, GS97] also fits into this category. In semi-Passive replication [DSS98] the primary-copy is selected automatically in the context of a variant of consensus and the update form the value that is decided. The Pronto replication protocol [FP01] replicates high-level transactions and uses the total order broadcast protocol to detect the case of two concurrent primaries. Some techniques send even less than one message per transaction, transactions are grouped in batches called *epochs* and backups install those changes by batches [GMP90] – those techniques are, by design, 1-safe.

Requirements. In the case of passive backups, as long as the transaction changes are installed in the same order as in the primary, the backups will consistently reflect what has happened at the primary. Thus, if the primary sends changes in FIFO order and is producing correct histories, so do the backups. The FIFO broadcast must also be uniform, as the broadcast ensures atomicity.

This also holds in the case of active backups as long as transactions only access data for which the executing site is the primary. Care has to be taken if transactions are also allowed to read data for which the executing site is not the primary or if transactions are distributed (i.e., they update data of different primaries). In this case, the scenario is similar to that in update everywhere server architectures and it is not enough that primaries send changes in FIFO order but a total order is necessary.

Discussion. Lacking a voting phase, this type of protocols are naturally cold-standby since the primary has no way of waiting for the secondaries to apply the changes. Depending on the guarantees offered by the communication system, techniques in this category can be 1-safe or 2-safe (see Chapter 4).

If the backups are passive, that is, they do not do anything but installing the changes sent by the primary, determinism simply requires to install the changes in the order in which they arrive from the primary. If the backups are active and are executing transactions on their own behalf, then there must be some rules to prevent inconsistencies. These rules can be summarised as follows: the local serialisation order cannot contradict the order in which the remote transactions arrive. In this case, strategies are similar to those described in Section 3.2.1.A.

Summary. The requirements for this technique are, therefore, as follows: The communication primitive must guarantee FIFO order for passive configurations and total order for active configurations.

3.2.2.B Primary-Copy – Constant Interaction – Voting Techniques

Description. The introduction of a voting phase allows us to ensure that both the primary and the backups install the updates. While 2-safe property can be ensured without a voting phase (see Chapter 4), having a voting phase is the traditional way to enforce the 2-safety property. Voting also offers flow control and thus hot-standby behaviour: the system must wait for all replicas to be ready to install the change of a transaction before committing the transaction. Therefore, a replica cannot be left behind (see Section 2.4.3). The protocol is the following (Figure 3.7):

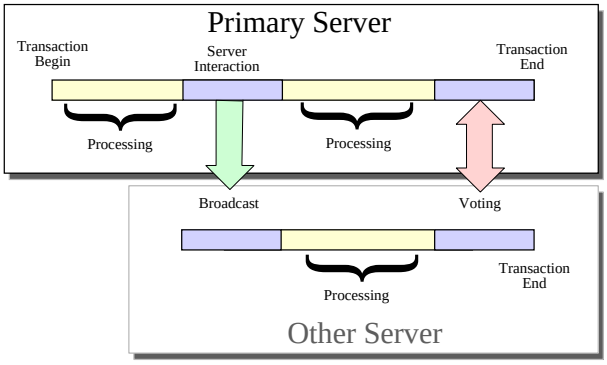

Figure 3.7: Primary-copy, constant interaction, voting

1. The transaction is executed at the primary.
2. When the transaction terminates, the corresponding log records are broadcast to all backups.

3. The primary initiates an atomic commitment.

4. The transaction is installed and committed at all sites.

References. Techniques in this category are typically 2-safe, hot standby primary-copy replication techniques [BGHJ92].

Requirements. Compared to *primary-copy– constant interaction – non-voting*, the voting phase makes it possible use a weaker communication primitive for broadcasting, as any discrepancies will be detected during the voting phase. This means that the FIFO broadcast might be non-uniform or best effort. However, since the primary waits until all backups have installed the transaction, the system is hot-standby. A weak voting scheme would be possible, but would simply imply first sending the transaction's log with a non-uniform FIFO broadcast, then sending a termination message with a uniform reliable broadcast, such techniques would be cold-standby. If both messages are merged, then the technique is equivalent to the non-voting technique.

Discussion. The nature of the broadcast primitive depends on what has to be achieved. In principle, since the voting phase is anyway done via an atomic commitment, there is no requirement for the broadcast primitive used when the transaction is sent to all backups (except FIFO). If there is any problem, the transaction will abort during the atomic commitment. The atomic commitment can be greatly optimised if used only as a synchronisation point and not to guarantee atomicity. It is an open research question how to balance these two aspects in terms of cost and overhead.

If the backups are active, the use of an atomic commitment allows to minimise the scheduling constraints: transactions can be executed without any determinism requirement and in any serial order. However, experience shows that minimising these constraints results in high abort rates. Thus, it is probably best to use total order broadcast and locally follow the delivery oder to avoid unnecessarily high abort rates.

Summary. Since aborting transactions at the backups due to serialisation problems is very inefficient, FIFO order for passive backups and total order for active backups is a reasonable requirement for this technique.

3.2.2.C Primary-Copy – Linear Interaction – Non-voting Techniques

Description. With constant interaction techniques, waiting until the transaction ends in order to propagate the changes means that the replicas will have trouble keeping up to date with the primary (hot-standby). The protocol could be faster if the backups can process transaction in parallel to the primary. In order to do this, the primary sends operations as they are executed, thereby allowing the backups to

start doing some work. If no voting phase is involved, the protocol is as follows (see Figure 3.8):

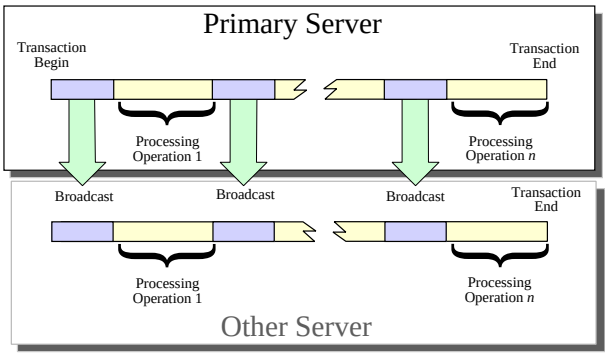

Figure 3.8: Primary-copy, linear interaction, non-voting

1. The transaction starts at the primary.
2. Read operations are executed locally.
3. The results of write operations are broadcast to the backups.
4. A termination message indicates the end of the transaction.

References Replication techniques that stream elements of the log as they are captured fall in this category [Sta95].

Requirements. Since the backups receive operations and not transactions, one has to be more careful about the order in which changes are installed. In the case of passive backups, if the primary produces correct histories and sends operations in serialisation order, then FIFO delivery is enough to guarantee correctness. In general, since what is being sent to the backups are log records and log records are produced in serialisation order, the primary does not need to make any extra effort to ensure this property. If the backups are active and transactions may access data across primaries, determinism is again achieved by relying on total order. By serialising according to this total order, overall correctness is assured – this is similar to what is described in Section 3.2.1.C.

The last message should be sent using a uniform FIFO broadcast. The reason why only the last broadcast needs to be uniform is the same reason than in Section 3.2.1.C for the update everywhere case: only the last operation leads to the commit. Since there is no voting, whether the protocol is hot- or cold-standby depends on whether the backups install the changes or they only save them to disk.

Discussion. Sending the operation as they are executed at the primary allows the backups to work in parallel but introduces a significant message overhead. Transactions typically have 20 update operations. Thus, to sustain a throughput of 100 transactions per second, the communication system must be capable of supporting a traffic of over 2000 broadcasts per second across the system. In practice, this is likely to be the biggest bottleneck when using this type of protocols.

Summary. This type of protocol has the following requirements: The communication primitive must guarantee FIFO delivery (passive backups) or total order (active backups) for the operations. The communication primitive must guarantee uniform delivery for the message containing the last operation.

3.2.2.D Primary-Copy – Linear Interaction – Voting Techniques

Description. As for techniques described in Section 3.2.2.B, the purpose of introducing a voting phase is to ensure hot-standby behaviour (Figure 3.9):

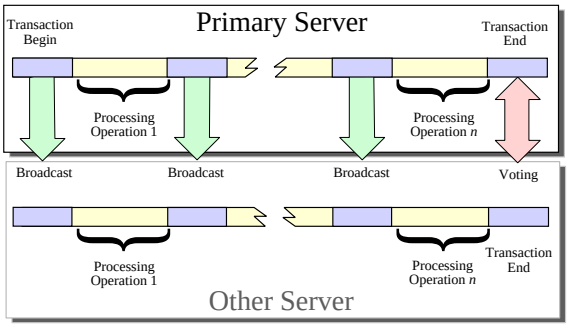

Figure 3.9: Primary-copy, linear interaction, voting

1. The transaction starts at the primary.
2. Read operations are executed locally.
3. The results of write operations are broadcast to the backups.
4. The primary starts an atomic commitment protocol.
5. The transaction is installed and committed at all sites.

References. This corresponds to the classical database replication scheme using a distributed transaction. This is very similar to the one described in Section 3.2.1.D.

Requirements. Compared to non-voting, correctness is not affected by the voting phase. Nevertheless, active backups are free to abort any transaction since they can propagate this decision during the atomic commitment phase. As the voting phase is now responsible for atomicity, the last FIFO broadcasts do not need to be uniform.

Discussion. As for primary, constant interaction voting techniques, the use of atomic commitment at the end of each transaction removes any requirements for the broadcast primitive. In fact, this protocol is very similar to traditional replication protocols, and the discussions in Section 3.2.1 in the context of voting techniques also apply here.

Summary. The requirements for this technique are minimal. Again, since aborting transactions because of serialisation problems at the backups is usually not acceptable, FIFO, respectively total order broadcast is a reasonable requirement for this technique.

3.3 Discussion

3.3.1 Overview of Requirements

	Strong Voting	Weak Voting	Non-Voting
Primary Copy	Atomic Commitment	FIFO Reliable Broadcast & Uniform Reliable Broadcast	Uniform FIFO Reliable Broadcast
Update Everywhere	Atomic Commitment	Total Order Broadcast & Uniform Reliable Broadcast	Uniform Total Order Broadcast

Figure 3.10: Requirements, according to classification

Figure 3.10 shows a summary of the requirements of the different techniques according to the classification. For clarity, only the constant interactions techniques are shown. The requirements for linear techniques are basically the same than the those for constant interaction techniques, simply instead of one broadcast, there are m broadcasts. Additionally, if the broadcast used for the constant interaction techniques needs to be uniform, then the corresponding linear interaction technique will only require the last broadcast to be uniform (see Section 3.2.1.C).

We also see that there is a symmetry between primary-copy and update everywhere. If an update everywhere technique requires a total order broadcast, the

corresponding primary-copy technique requires FIFO broadcast – all other requirements, reliability and uniformity stay the same. In the primary-copy scheme, the primary knows in what order messages will be delivered by the FIFO broadcast, and so can make sure that there will be no serialisation problem.

3.3.2 Server Architecture: Primary-Copy vs. Update Everywhere

While update everywhere seems a more desirable solution by promising load balancing and easier configuration, actual techniques in use nowadays are primary-copy techniques.

The first reason for this is simplicity. Primary-copy techniques can be implemented relatively easily, for instance by using high-level techniques like triggers and stored procedures [Sta94, Gol94]. In some cases, the primary-copy runs an unmodified version of the database, with a special process capturing updates and sending them to the back-ups [BT00].

Update everywhere does not necessarily distribute the load among sites. Since the data is replicated, all sites need to perform the updates anyway. This means that unless there is a significant amount of read operations in the overall load (read operations being local), the system might not scale up as more server nodes are added.

One way to improve the performances of update everywhere is to preprocess operations at one site and send the results to the other sites, that is, transform high-level transactions into low-level transactions [Kem00]. That way, the processing does not need to be done everywhere. Once such mechanisms are in place, update everywhere becomes a more attractive solution since it is more robust to failures and facilitates distributing the load among the sites.

3.3.3 Server Interaction: Constant vs. Linear

The number of messages exchanged per transaction is a key aspect of any replication protocol. As pointed out before, sending one message per operation can quickly lead to unacceptable traffic rates. In addition, these messages need to be processed at each site, which significantly increases the load. Finally, since operations arrive at different points in time, coordinating their execution so that the overall result is correct is much more complicated.

It is a good rule of thumb to say that the less messages exchanged per transaction, the better. For instance, protocols based on linear interaction in combination with update everywhere are largely infeasible in databases. It is exactly this type of protocols that have been heavily criticised in the database community as unrealistic [GHOS96]. In the primary-copy case, things are a bit different but the consequences of how many messages are exchanged are not negligible. In particular, sending all updates in one message at the end of the transaction can help to propagate the changes of only those transactions that actually commit.

Sending changes that will abort is useless and lowers performance: it uses network bandwidth to transmit unneeded messages which cause contention on the other servers. Those transactions will requests locks, and their operation will be logged.

Exchanging one message per transaction, however, introduces its own problems. Protocols that do so, work especially well for service requests where the data to be accessed is known in advance. In this case implementation is straightforward and abort rates are small. Those techniques can even be optimised in order to handle the network protocol and transaction processing in parallel, yielding better response times [KPAS99].

However, for ordinary transactions, some form of optimism must be used to first execute the transaction at the delegate server and then determine the serialisation order. If the conflict rates are high, this optimism might result in high abort rates.

Also, in those techniques the delegate needs to wait for the message to reach all servers before being able to decide on the outcome of the transaction. This means that write locks will be held during some time, which might lead to high lock contention and increase response time.

3.3.4 Transaction Termination: Voting vs. Non-voting Techniques

Non-voting techniques are more demanding in terms of determinism requirements than voting techniques. With non-voting protocols, each server must independently guarantee the same serialisation as that of other servers. The typical way to do this is to use the total order as a guideline. In general, if two transactions conflict, their serialisation order will be that indicated by the total order. Depending on the protocol, sites need to known different things in order to ensure global correctness without voting. There are protocols where the whole transaction (read operations included) is sent. In these protocols, each site performs the equivalent of global scheduling for the whole system and, as long as this scheduling is deterministic, correctness is guaranteed. This determinism can be implemented by following the total order to serialise transactions.

In other protocols, only update operations are sent. In these protocols one has to be careful about read operations since they are not seen by all sites and can alter the serialisation order. This means that only the servers that know the read operations can decide if the execution is serialisable. If no voting is allowed, then transactions that are known to all sites take priority and, in case of conflicts, only local transactions can be aborted (local meaning those not seen by all sites).

In terms of voting techniques we have considered two possibilities, one of them is based on atomic commitment and another based on a confirmation message sent by the delegate or primary-copy to indicate whether the transaction can be committed or must be aborted. The confirmation message is needed when only the delegate server (or the primary-copy) of a transaction can unilaterally decide on the outcome of the transaction. However, remote sites must still behave determin-

istically in such a way that they must be able to obey the commit/abort decision of the delegate server.

When atomic commitment is used, each server can reject any transaction thus relaxing the determinism requirements since there is always a chance to resolve things during the atomic commitment. Unfortunately, it has been shown that in these cases, the coordination overhead is much higher, and according to [GHOS96], conflict, abort and deadlock rates can quickly become a bottleneck. Additionally, when voting is also used to provide atomicity, it can only take place when all sites have completely executed the transaction. This means that the delegate server waits for the slowest of all replicas to finish processing before returning the result to the client, increasing transaction response times considerably.

Chapter 4

Recovery and Fault-Tolerance Issues

If we do not succeed, then we run the risk of failure.

Dan Quayle

One of the goals of replication is fault tolerance – if certain replica crash, the database can still be available. In this chapter, we examine the fault tolerance guarantees that can be obtained by basing database replication on group communications.

The basic safety criteria in the database world are 1-safety and 2-safety [GR93]. Traditionally, recovery of systems based on group communications is based on the view change mechanism. We show in this chapter that this mechanism is not adequate for building 2-safe replication and show the adequate recovery mechanism. We also introduce a new safety criterion, stronger than 1-safe, but weaker than 2-safe, which we call *group-safe*. This criterion is more suited to group communication-based replication and permits implementations that promise increased performance.

This chapter is structured as follows, Section 4.1 introduces the 1-safe and 2-safe safety criterion. Section 4.2 introduces view-based group communication systems. Section 4.2.1 explains the current shortcomings for building a 2-safe replication technique. Section 4.2.2 shows that even if those shortcomings were corrected, the basic recovery mechanism in view-based system cannot be used to build a 2-safe replication technique. Section 4.3 discusses the check-pointing mechanism needed for 2-safe replication. Finally, Section 4.4 introduces a new safety criterion and discusses its implications.

4.1 Safety Criterion

There are three safety criteria, called 1-safe, 2-safe and very safe [GR93]. When a client receives a message that a transaction committed, durability ensures that the effects of this transaction are preserved in stable storage. However, depending on the safety criterion, the transaction might be lost in case of crash.

1-safe If the technique is 1-safe, when the client receives the notification of t's commit, t has been committed on the delegate or the primary server of t.

2-safe If the technique is 2-safe, when the client receives the notification of t's commit, t is guaranteed to eventually commit on all servers.

very safe If the technique is very safe, when the client receives the notification of t's commit, t is guaranteed to *have* committed on all servers.

Each safety criterion has a different tradeoff between the safety and the availability of the system: 1-safe replication ensures that transactions can always be accepted, but some transactions might get lost in case of a crash. A very safe system ensures that a transaction *has* committed on all servers, but this means that a single crash renders the system unavailable. This last criterion is not very practical and most systems are therefore 1-safe or 2-safe.

The distinction between 1-safe and 2-safe replication is important in case of a crash. If the technique is 1-safe, transactions might get lost if one server crashes and another takes over. If the technique is 2-safe, no transaction can get lost, even if all servers crash.

All those safety constraints implicitly assume a read-one, write-all (ROWA) policy. Because of this, those safety constraints are difficult to map to general quorum-based replication. In a quorum system, a given transaction might never commit on a certain replica. Because of this, quorums will not be considered in this discussion.

4.2 View-Based Recovery

4.2.1 Existing Systems

Database replication based on group communication usually considers that the group communication is defined in a dynamic crash no-recovery model [HAA99b, KA00a, Ped99, PF00]. One reason for this is that most current group-communication toolkits [BJ87, MFSW95, vBM96, DM96, MMSA$^+$96] are based on this model. Recovery in this model is based on the view mechanism (see Section 2.2.3.B). When a crashed replica b recovers, a *view-change* occurs. The group communication system requests the state from one replica a to transfer it to replica b. This action is called a *state-transfer*. In other words, the group communication system forces a checkpoint on one replica (a), and does a roll-forward of the recovering replica (b) to this checkpoint.

The application of this recovery mechanism to replicated databases is described in [KBB01]. Some optimisations are needed to avoid transferring the state of the whole database each time a database server recovers, but conceptually the recovery mechanism stays the same.

There are two issues with view-based system. First view-based systems cannot tolerate a total crash: if all server crash, the group communication layer blocks and the system cannot recover. This alone precludes the use of current group communications for building 2-safe replication techniques. 2-Safety places no bounds on the number of crashes, e.g. a 2-safe technique must be able to tolerate an arbitrary number of servers crashing. Second, even if a view-based system *could* tolerate a total crash, the recovery mechanism would still be insufficient to ensure 2-safety. This problem is discussed in the next section.

4.2.2 Roll-Forward Recovery

The main issue with state transfer recovery scheme is that it is insufficient to build a 2-safe replication, regardless of the fact that the group communication layer can tolerate a total crash. When a client is notified of the commit of transaction t, the only guarantee is that t was committed by the delegate. The use of group communication does not ensure that t *will* commit, but merely that the message m containing t was *delivered* in the view. If t is not committed because of a crash, t's effects must be applied by the recovery mechanism.

If a crash occurs, a replica can only recover if there is another replica that is up with an available state. This is basic assumptions of roll-forwarding: one replica must be available to do a state-transfer. If no replica has a consistent state to make a state transfer, or if the replicas with a consistent state are crashed, then recovery is impossible. For this reason, group communication systems that use view change mechanism cannot be 2-safe.

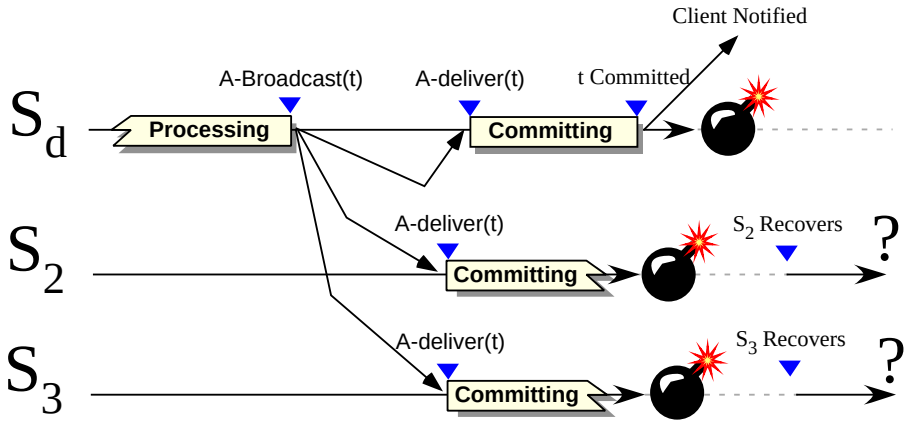

Figure 4.1: Unrecoverable failure scenario

To illustrate this problem, consider the following scenario (Figure 4.1): transaction t is submitted on the delegate replica s_d. When t terminates, s_d sends a message m containing t to all replicas. The message m is sent using a total order broadcast. The delegate s_d delivers m locally, commits t and confirms the commit to the client, and finally crashes. So s_d processed m and then crashed (t did commit). All other replicas (s_2 and s_3) deliver m but crash before committing t. The system cannot rebuild a consistent state that includes t's changes. So the technique is not 2-safe.

This failure scenario illustrates the issue with a non-voting replication technique, but similar scenario exist for other replication schemes, including voting replication schemes. Does this mean that voting replication is not 2-safe? No, because voting replication is usually done using atomic commitment protocols that are not based on the dynamic crash no-recovery model. For instance, the 2PC protocol includes logging phases, so if a crash occurs during the protocol, the decision of the protocol can be played back upon recovery.

4.2.3 Conclusion

It is impossible to build a 2-safe replication technique using a group communication system based on the crash no-recovery model. The reason is that the group communication system only guarantees that messages are *delivered* within a view, but offers no guarantees that the application will be able to *process* those messages. Moreover, the fact that a message is delivered does not mean that it is processed by the application. Recovery schemes [Hol01] based on this assumption are simply incorrect. To build 2-safe replication, we need another model for the group communication system.

4.3 Roll-back-based recovery

In order to build a 2-safe database replication scheme, we need to solve the issues related with view-based recovery. 2-safety is an end-to-end property, so to ensure 2-safety, we need the group communication system to ensure not only that messages are delivered, but that they are processed by the application (the database). In order to be sure that recovery is always possible, recovery should be based on roll-backs, not roll-forwards.

In order to get this, we need to synchronise the application layer and the group communication layer. As outlined in Section 2.2.3, both the application layer and the group communication layer contain some state information. One of the key issues when recovering a database replication technique based on group communications is the synchronisation between the application layer (the database) and the group communication system. Transactions that were executing at the time of the crash lead to problems. They might have been delivered by the group communication layer, but not yet committed by the database. For one layer, they have

been processed, for the other they have not. Upon recovery, application and group communication layer need to synchronise to restore those transactions. Because of this, we need to define very precisely the relationship between the group communication layer and the application layer.

4.3.1 Inter-Layer Messages

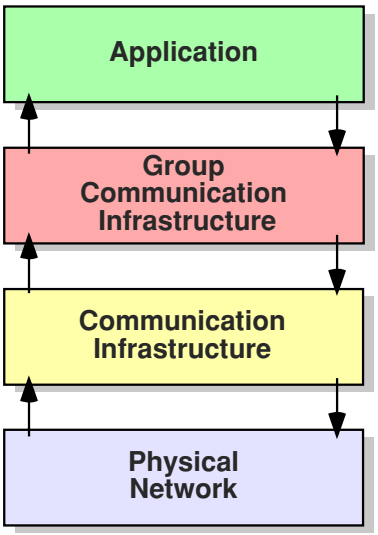

Figure 4.2: Protocol stack

Figure 4.2 shows the classical layered model that we consider. The top layer is the application, with the group communication system underneath, then the communication infrastructure, and finally the physical network. Each layer relies on the layers underneath to implement the services it offers to the layer above.

While inter-process communications are well specified, inter-layer communication, and more specifically communications between the group communication layer and the application layer are only defined partially, typically in the form of function calls – which often hides or confuses the flow of control. Many optimisation of group communication primitives also need to change those interfaces [DSS98, BFG01].

We express the communication between the group communication layer and the application layer as *messages*. When the application executes a certain primitive, it sends a message to the group communication layer. When the group communication layer delivers a message or a decision to the application, the group communication sends a message to the application. So when for instance a replica A-delivers a message m, message m is sent to the application by the group communication layer.

This way, we can model the inter-layer (intra-process) communication in the

same way than inter-process communication. The main difference is that all layers share the same process, and therefore fail at the same moment. Channels for inter-layer communication are reliable (no message loss, except in case of crash).

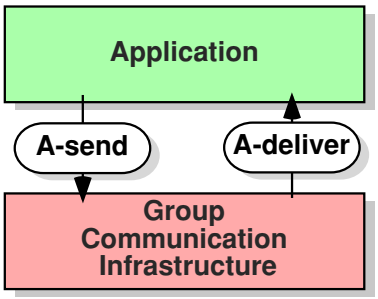

Figure 4.3: Messages exchange for total order broadcast

Figure 4.3 shows for instance the messages exchanged between the application and group communication layers for the total order broadcast protocol. The application wants to A-send a message, later on, this message is A-delivered to the application.

4.3.2 Inter-Layer Ack Messages

The specification of group communication primitives ensures that messages are sent from the group communication to the application in precise circumstances and precise order. Yet, the specification does not guarantee that the application will be able to handle those messages. If the application cannot handle a message m because of a crash, we say that the delivery of m was *unsuccessful*. The application must roll-back before the delivery of m, and m should be delivered again ("replayed") [RR00].

To ensure this, the group communication system needs to know when the application has finished processing m, that is, when the delivery of m was *successful*. This is done by having the application send a message $ack(m)$ to the group communication layer when the processing of m is finished. This mechanism is similar to acknowledgement messages used in inter-process communications.

We assume a well-behaved application, that is, when the application receives message m from the group communication layer, it will send $ack(m)$ as soon as possible. No application waits forever before sending $ack(m)$. If a crash occurs, and the group communication layer did not receive the message $ack(m)$, then m should be sent again to the application. So after each crash, the group communication layer "replays" all messages m whose ack message $ack(m)$ where not received from the application. By replaying messages, the group communication layer ensures that, if the process is eventually stable (non-red), then m will eventu-

ally be successfully delivered.[1]

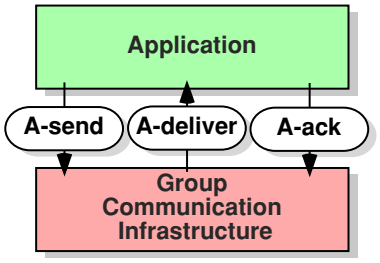

Figure 4.4: Messages exchange for total order broadcast including *ack* message

Figure 4.4 shows the exchange of messages for an total order broadcast including the *ack* message. First, the application sends a A-send message to the group communication layer. The A-deliver message is received by the application. The application processes it, then sends back an A-ack message to signal that it *successfully delivered* m.

Note that the group communication layers needs to garbage collect certain resources, messages stored in memory or in stable storage, identifiers etc. Once the group communication layer receives $ack(m)$, then all resources associated with m can be reclaimed. A mechanism to minimise log usage for total order broadcast based on a similar idea is presented in [Boi01].

4.3.3 2-Safe Replication

So what are conditions needed to build a 2-safe database replication scheme based on group communication? In order to be 2-safe, a replication scheme must ensure that, eventually, all transactions will be terminated correctly on all non-red replicas. If the decision to commit transaction t is delivered in message m, then t will be committed when m is successfully delivered. So if the group communication layer ensures that all messages are eventually successfully delivered on all non-red replicas, even when all replicas crash (no green replicas), then the replication technique will be 2-safe. So in summary we need the following conditions:

- The group communication layer can tolerate that all processes crash ($green = 0$).
- The group communication layer can replay messages that where not successfully delivered on all non-red processes.

[1]The specifications of group communication primitives described in Section 2.2.4 need to be refined, as the integrity constraint guarantees that messages are delivered at most once. When messages are replayed, a message might be delivered multiple times, but will be *successfully* delivered at most once.

Those properties can be achieved by relying on the crash-recovery model with stable storage (see Section 2.2.3.D). Formally, the group communication layer simply needs to log message m before sending it to the application. When the group communication layer receives $ack(m)$, m can be deleted from stable storage. In practice, the logging of m can be merged with other logging operations in the group communication layer[2] and so cause no additional performance penalty.

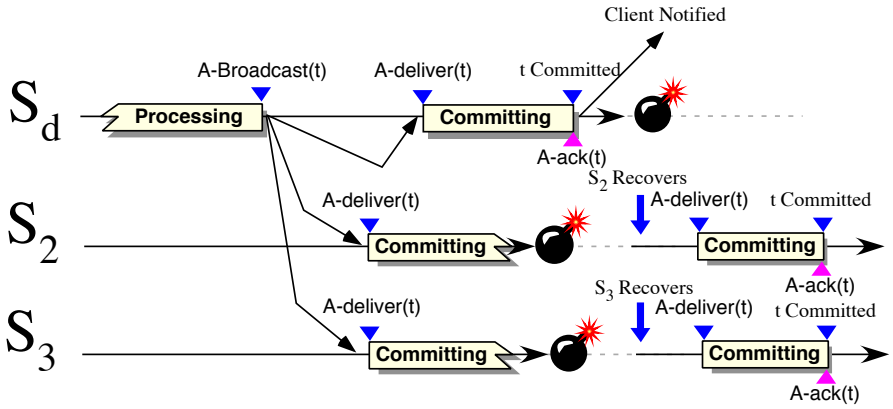

Figure 4.5: Recovery with message replay

Figure 4.5 shows the same scenario than in Figure 4.1, but with a group communication layer that can survive a total crash and replays messages. When the crash occurs, replicas s_2 and s_3 are rolled back to before the delivery of m and the delivery of m is restarted and eventually the delivery of m is successful, and thus t is committed. This way, 2-safety is ensured.

4.4 Group-Safe Replication

4.4.1 Group-Safety

We have shown in Section 4.2 that techniques based on the dynamic crash no-recovery model cannot be 2-safe. Such techniques are 1-safe: when the client is notified of t's commit, t did commit on the delegate server, but not on the other servers. Traditional 1-safe techniques can lose transactions if one server (the primary/delegate) crashes [GR93]. This is not the case for techniques based on group communications: if the primary/delegate server crashes, the broadcast mechanism ensures that the transaction reaches all servers and is committed there. Transactions can be lost only if enough servers crash to compromise the group, for instance if the model is asynchronous with imperfect failure detectors and a majority of servers crash, transactions might get lost (see Section 4.2.2). So group communication

[2]Group communications in the crash-recovery model where all processes can crash implies stable storage, and hence messages are logged.

based replication schemes enforce a safety level stronger than 1-safety, but weaker than 2-safety. We call this safety level *group-safety*.

A replication technique is *group-safe* if, when a client receives confirmation of a transaction's commit, the transaction has been delivered on all (available) replicas. Traditionally, database systems rely on stable storage to ensure durability: once a transaction is committed, it cannot be lost. If stable storage does not fail, durability is ensured. Group-safety relies on the group of replicas. If the group does not fail, for instance if there is always a majority of non-crashed replicas, then durability is ensured. Notice that group safety does not specify if the transaction actually committed on any replica. A client might be notified of a transaction t's commit before t actually committed on any replica.

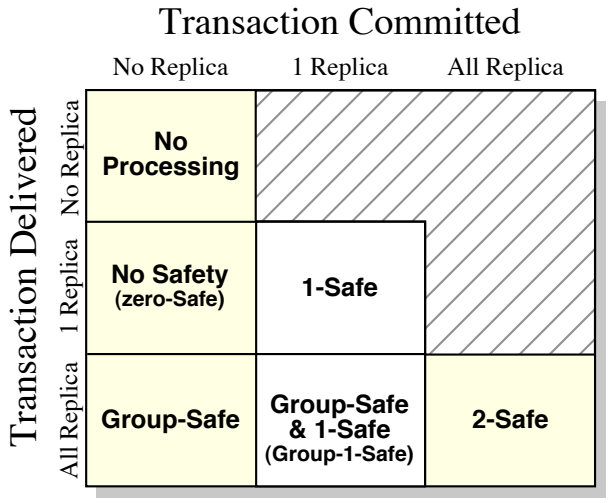

Table 4.1: Summary of different safety levels

The relationship between group-safety and the safety criterion described in Section 4.1 are summarised in Table 4.1. We use a classification based on two criteria: the number of servers that have delivered the transaction (vertical axis), and the number of servers that have committed it (horizontal axis). Depending on the safety level, a transaction t can be *delivered* on a certain number of replicas (none, one, all) and *committed* on a certain number of replicas (none, one, all). A transaction cannot be committed on a site where it was not delivered, so the part of the table where $committed > delivered$ is grayed out. For each position in the remaining table, the corresponding safety level is given and described below:

No Processing When a transaction is delivered on no replica, no processing is possible.

No Safety In this case, the client is notified when transaction t was delivered and executed on one server s_d, but did not yet commit. No safety is enforced. If

s_d crashes before t's writes were flushed to sable storage, then t will be lost. We call this safety *zero-safe* replication.

1-Safe In a *1-safe* system, the client is notified when transaction t is delivered and committed on one server only: s_d. If s_d crashes before t is sent to the other servers, then t might get lost. This situation occurs because the system can accept new transactions that conflict with t. As no server except s_d knows about t, the only alternative to losing t is to block all new transactions until s_d recovers.

Group-Safe The client is notified when a transaction is delivered on all available servers, but did not commit on any servers. If the group fails – typically when more than f servers crash – then t will be lost ($0 < f < n$, the exact value of f depends on the model of the group communication system, see Section 2.2).

Group-Safe & 1-Safe The client is notified when transaction t delivered on all servers, and did commit on one server s_d. The system is both group-safe and 1-safe, so we call this safety level group-1-safety. Transaction t might be lost if f servers **and** the server s_d crash. Most proposed database replication strategies based on group communication fall in this category.

2-Safe The client is notified when a transaction was delivered and committed on all available servers. A transaction cannot get lost.

If we consider the number of crashes that can be tolerated, we have basically three safety levels (Table 4.2). Zero-safe and one-safe replication can tolerate zero crashes, i.e., one single crash can mean a lost transaction. Group-safe replication can tolerate f crashes and 2-safe replication can tolerate n crashes (n is the number of servers, $0 < f < n$).

Number of Crashes	Safety Constraint
0 crashes	zero-safe, 1-safe
f crashes	group-safe, group-1-safe
n crashes	2-safe

Table 4.2: Safety constraints and number of crashes

The difference between zero-safe and 1-safe replication is the behaviour in case of crash of the primary/delegate. If a crash occurs while transaction t is processed, t will be lost if the technique is zero-safe. If the technique is 1-safe, t has committed on one server s_d, and is therefore preserved on stable storage. So t can be restored if the system waits for the s_d to recover. So if no conflicting transaction is accepted until this moment, t can be restored. In zero-safe replication, t is lost because of a crash, in 1-safe replication, t is lost because of a crash **and** a conflict.

The difference between group-safe and group-1-safe replication is similar. If all servers crash (total crash) while t is processing, and the replication scheme is group-safe, then t will be lost (it was not stored in stable storage). If the replication scheme is group-1-safe and all servers crash while t is processing, t was committed to stable storage on server s_d. This means that t can be restored **if** the system waits for s_d to recover.

The advantage of 1-safe over zero-safe and group-1-safe over group-safe is the same: the possibility to restore transactions from stable storage. For both 1-safe and group-1-safe there is a tradeoff between availability (accepting new transactions) and safety (potentially violating the ACID properties by committing transactions that conflict). Waiting for the primary server to recover can be an acceptable alternative to losing transactions in a primary-copy setting, where the number of servers is small and the primary can be restored in a short time. In an update-everywhere setting, where all servers can act as delegates, this is not an acceptable option: it means that while the system can withstand a total crash, it can only recover from such a crash by recovering all replicas.

So in a practical update-everywhere setting, the difference between group-safe and group-1-safe replication in terms of availability in fault tolerance is negligible. Group-1-safe replication offers a possibility of restoring transactions from stable storage, but in practice this possibility cannot be used without lowering the availability of the system.

4.4.2 Group-based durability

Most group communication based replication techniques are group-1-safe. They enforce both 1-safety and group safety, but the practical difference between both safety criteria is very small. What would be the advantages of enforcing only group-safety and renounce to 1-safety?

In a non-replicated setting, writing to stable storage is necessary to ensure *durability*. The most expensive part of transaction processing is composed of *force-writes*, i.e., writing data in stable storage. A transaction cannot commit before its changes are stored in some form on stable storage. Read operations can be optimised using mechanisms likes caches, but force-writes are generally very expensive.

The key idea of group safe replication is that durability is not ensured by stable storage anymore, but by the *group* (i.e., the group of servers). In a non-replicated and in a 2-safe replicated database, durability is ensured by stable storage. As long as the stable storage does not fail, transactions cannot get lost and durability is guaranteed. In a group-safe replication scheme, durability is ensured by the group: if the group does not fail (less than f servers fail), then no transaction will be lost and durability will be guaranteed.

A group-safe replication technique does not require any replica to actually have finished the commit of t before responding to the client. So the response might be sent to the client before any actual force-write operations are executed. Transfer-

ring the responsibility of durability from stable storage to the group makes sense a lot of sense. Sending a message through the network is faster than writing the same data to disk [DWAP94].

4.4.3 Group-safe replication and lazy replication

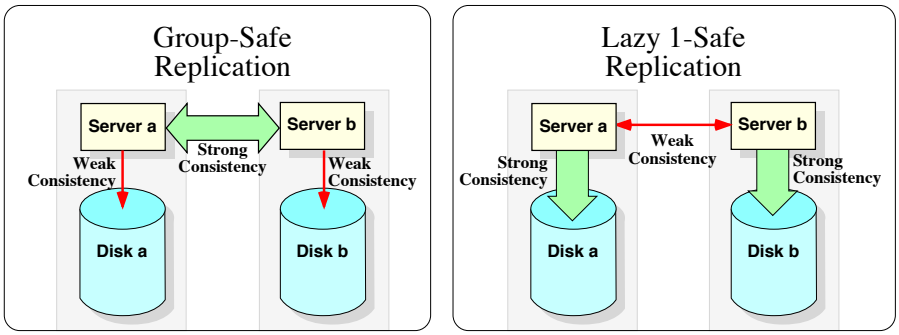

Figure 4.6: Group-safe replication and lazy replication

On a conceptual level, group-safe replication is the complement to lazy replication. Both approaches try to get better performance by weakening the link between some elements of the system. Figure 4.6 illustrates this relationship. Group-safe replication relaxes the link between server and stable storage: when a transaction t commits, the state in memory and in stable storage might be different (t's writes are not committed to disk). Lazy replication relaxes the link between replicas: when a transaction commits, the state in the different replicas might be different (some replicas have not seen transaction t). Both approaches relax the link whose synchronisation cost is deemed to expensive.

The main differences between both cases are the conditions that lead to a violation of the ACID properties. In an update-everywhere setting, a lazy technique can violate the ACID properties even if no failure occurs. On the other hand, a group-safe replication will only violate this ACID properties if the group fails (for instance if a majority of all servers crash).

Group-safe replication also has one large advantage over lazy replication in cases where the number of replicas n is large. The main problem of lazy replication lies in reconciliation, as the number of servers grows, the chances that two transaction originating from two different sites will conflict grows. So the chances that the ACID properties will be violated grows with the number of servers. Group-safe replication does not have the problem of reconciliation – on the other hand the ACID properties might get violated if f servers crash. So when n increases, the number of failures needed to violate the ACID properties increases also. If we assume that the probability of f failures occurring at the same time decreases as f increases, then the higher the number of replicas, the lower the chances of having

a lost transaction. So the chances that something *bad* happens increases with n for lazy replication, and decreases with group-safe replication.

4.4.4 Building a Group-Safe Replication Technique

Most group communication-based replication techniques described in the literature are both 1-safe and group-safe. They can be transformed in group-safe only replications techniques quite easily. Group-safe replication basically means that all disk writes can be done asynchronously (outside of the scope of the transaction) thus enabling optimisations like write caching. Typically, disk writes would not be done immediately, but periodically. Writes of adjacent pages would also be scheduled together to maximise drive throughput.

For increased safety, a group-safe database replication can choose to dynamically change its policy regarding writes. In normal operation mode, writes are asynchronous, and the technique is group-safe. In a problematic situation, for instance when a certain number of server are crashed, writes are synchronous and the technique becomes group-1-safe.

4.5 Conclusion

Group communication-based database replication is a promising approach to database replication, but existing safety properties are not adapted to quantify those new techniques. Traditional safety criteria like 1-safety and 2-safety implicitly assume that durability is ensured by stable storage alone.

The group communication infrastructure to build true 2-safe replication techniques exists in theory, but practical implementations are not available yet. Implementation could either be based on the crash-recovery model, or be an additional layer on top of existing group communication toolkits. In the second case, a second explicit round of messages would be needed to signal that messages have been logged to stable storage [Kei94].

Group communication-based 2-safe replication techniques are possible and would address some of the issues of classical 2-safe replication techniques like deadlocks [GHOS96, HAA00]. Yet the network protocols needed to build 2-safe replication need to have access to stable storage. If stable storage is to be implemented using disk, the resulting protocol will certainly be as slow as traditional 2-safe techniques.

Most group communication-based replication techniques that were proposed recently are not 2-safe, but offer more than 1-safety. Those techniques are both 1-safe and *group-safe*. By removing the 1-safety criterion, we move the responsibility of durability from stable storage to the *group*. While executing writes outside of the scope of a transaction is a known way to improve the performance of transaction processing systems, this generally implies not enforcing durability. Group-safe replication enforces durability and offers the same availability and fault-tolerance

that group-1-safe replication, with increased performance.

Simulations show (see Section 5.3.5) that the performance of group-safe replication is very good, and offers an alternative to lazy replication with better performance and better consistency. This shows that good performance can be obtained for database replication without sacrificing consistency.

It is interesting to note that for both approaches described in this Chapter, the functionality of the log is transferred to the group communication system. For 2-safe replication, the group communication layer logs messages before delivering them, so transactions do not need to be logged at the application level, as they are already logged by the group communication system. In the case of group-safe replication, safety is enforced by the group, and writes are done asynchronously, so logging does not make sense. In fact the group acts as a replicated, volatile log (messages are kept by all replicas, but not in stable storage). This shows clearly that to build an efficient replicated database system, the logging facility and the group communication layer need to be carefully integrated.

Chapter 5

Performance Comparison

There is more to life than increasing its speed.

Mahatma Gandhi

This chapter presents a quantitative comparison between different replication techniques. This comparison is performed using a simulator. Other simulation studies that investigate techniques based on group communication have already been described in the literature [HAA99b, Kem00]. The present simulation includes a more detailed model of the network layer and also covers replication algorithms that have not been simulated before.

The presentation of the simulation results follows the structure of the classification presented in Chapter 3. Except for one class (linear interactions, non-voting) which has been shown to perform badly [HAA99b], we simulated one technique per class in the update-everywhere category. Three group communication-based replication techniques have been simulated along with classical replication strategies: lazy replication, distributed locking and primary-copy replication. One of the group communication-based techniques, called the database state machine (or certification based replication), was never compared in simulation with other replication techniques. Additionally, the performance gains realised with some optimisations in group communication-based replication techniques are also evaluated.

This chapter is structured as follows: Section 5.1 describes the architecture of the simulator, and presents the different replication techniques. Section 5.2 describes the general settings used for the experiments and Section 5.3 describes the different experiments and their results. Section 5.4 discusses the results and compares them to other simulation and performance measurements in the literature.

5.1 The RD-sim Simulator

All experiment described in this chapter were performed using a discrete event simulator: RD-sim. The simulator is written in C++ and relies on the C-sim discrete event simulation engine [Mes94]. The goal was to have a detailed simulation to get an understanding of how different parameters of database replication interact. The resulting simulator is over 13'000 lines of code long.

The overall simulator can be roughly divided in two conceptual parts: the clients and the servers. The clients represent the source of transactions: they generate transactions according to certain parameters, send them to servers and collect end-to-end performance data. The servers implement the whole replicated database logic, including the local database, the group communication system and the replication strategies. The simulation concentrates on low-level aspects. High-level issues, like transaction parsing and optimising are not considered.

5.1.1 Server Structure

The architecture of the servers follows the logical structure outlined in Chapter 2, Figure 2.1. Each replica is both a network node with a group communication stack and a local database system. The simulator is therefore structured in four large modules, as illustrated in Figure 5.1. The four modules are:

1. The low-level machine module.
2. The communication module
3. The database module
4. The database replication module

Both the communication module and database module run on top of the low-level machine module. They implement the group communication infrastructure and a local non-replicated database, respectively. The database replication module represents a replicated database and is implemented on top of the group communication module and the database module. All modules have been implemented as one or more C++ classes, which export the relevant functionality.

5.1.1.A Machine Module

The Machine Module represents the hardware of one machine. Machine Modules are only used to simulate servers. There is one instance of this module for each server in the system. Each server machine is simulated using two basic resources: CPU and disks. Those resources are used by other high-level modules of the simulator. The CPU resources model the processing units. The disks resources are used by the database module. Basic input/output operations use both the CPU and the disk resources.

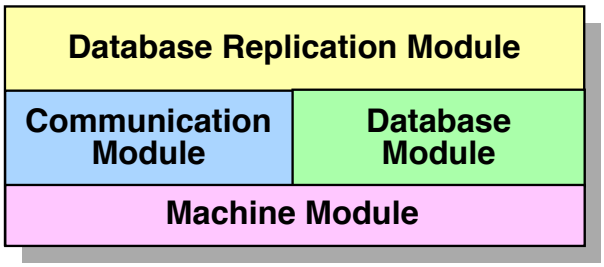

Figure 5.1: General simulator architecture

Theses low-level resources (CPUs and disks) are simulated as C-sim resources. High-level operations, like the execution of a network protocol, or that of a transaction operation, are implemented and executed by the simulator. Other simulations of replication techniques based on group communication [Kem00, HAA99b] simulate high-level protocols like total order broadcast as a single operation. This hides the complexity of the group communication sub-system and ignores the interactions between communication system and database system: for instance the CPU is used both for processing transactions and sending messages. Simulating the system at a finer granularity makes it possible to understand what role low-level resources play in the performance of different replication techniques. This way, we can see the influence of the network performance on the overall system.

5.1.1.B Communication Module

The Communication Module models all network interactions. There is one instance of this module for each server. At a low level, both point-to-point and multicast messages are modelled. High-level group communications primitives described in Section 2.2.4 are implemented on top of those low-level messaging facilities. The use of high-level primitives like total order broadcast will therefore result in a simulated execution of a total order broadcast protocol using low-level messages. The algorithms are simulated in failure-free runs – the most common case in normal operating conditions.

The Communication Modules relies on two kinds of simulated resources: the CPU and the network. The CPU is the resource exported by the Machine Module. The network is a resource shared by the Communication Modules of all servers: it represents the network between the servers.

The sending of message is modelled in three steps: first the outgoing message is processed on the sending node, then the message transits through the network, and finally the message is processed on the receiving node [UDS00]. The resources queues are handled by a FIFO policy. This means that three resources are involved in simulating the sending of one message: the CPU resource of the sending machine, then the network resource and finally the CPU resource on the receiving

machine. This way we can model network and CPU contention between messages, but also contention between the communication system and the database system.

5.1.1.C Database Module

The database module simulates a single database system. This modules includes a lock manager and an I/O manager. The lock manager offers lock queues for each item in the database. The lock queues can be used to enforce strict 2 phase locking (2PL), but the locking manager also supports other locking operations, like atomic locking for a transaction or "force-locking", that is, inserting locks in front of the lock queue. Those variants are typically used for non-voting replication techniques. The I/O manager handles operations to read and write items of the database. Data items are distributed on the different disks of the machine (each disk holds a partition of the data). The I/O manager also simulates the cache system.

5.1.1.D Database Replication Module

This module represents the database replication strategy. There is one instance of this module for each server. Depending on the replication strategy, a different implementation of this module is used. Each replication scheme is represented by an concrete subclass of the abstract Database Replication class.

We implemented one strategy for each update everywhere category, except for non-voting update everywhere with a constant number of interaction (Section 3.2.1.C). This category was never considered as interesting for replicated databases. One algorithm in this category was simulated in [HAA99b], and its performance was significantly worse than other group communication-based techniques.

We implemented three update everywhere replication techniques based on group communications, and the distributed locking replication technique. Additionally, we implemented two primary-copy techniques (cold standby and lazy) and one non-replicated technique to get a reference for relative performance.

The implemented techniques are the following:

No Replication. A special setting lets the system run a single, non-replicated database using standard 2 phase-locking (2PL). This is useful for comparison purposes.

Active Replication. This technique fits into the update-everywhere constant interaction, non-voting category with an early point of determinism (Section 3.2.1.A). The point of determinism is at the beginning of the transaction. Transactions are simply forwarded to all servers using a total order broadcast. All servers process the transaction in a deterministic way. This technique requires static transactions (see Section 2.4.4).

Optimistic Active Replication This technique is a variant of the active replication scheme based on an optimistic assumption on the network (spontaneous ordering, see Section 2.4.6). Transactions are delivered before the total order is determined and processing starts using this tentative order. When the total order is determined, the system checks if the execution according to the tentative order respects one copy serialisability. If this is not the case, the execution is aborted and restarted according to the definitive total order. This algorithm is presented in [KPAS99] under the name Fine Granularity Locking (FG-locking).

Certification. This technique is described in [PGS99] and [Ped99] under the "database state machine" name. It fits into the update-everywhere constant interaction, non-voting category (Section 3.2.1.A) like active replication. The difference between active replication and certification replication is the position of the point of determinism. In the certification technique, the point of determinism is late (last operation before termination) instead of being at the beginning of the transaction. Transactions are executed on the delegate. At commit time, the transactions are sent to all replicas using a total order broadcast. A certification mechanism checks if the transaction might conflict with other transactions that are currently executing. If this is the case, the transaction is aborted by all replicas. The certification test uses a conflict list that contains a list of transactions recently delivered that might not have committed on all servers [Van00].

Group-Safe Certification. This technique is the group-safe variant of the certification technique (see Section 4.4). The technique is similar to the certification technique, but uses asynchronous write operations.

Weak-Voting. This technique is described in [Kem00] under the name SER-D. It fits in the update everywhere, constant interaction voting category (Section 3.2.1.B). It is similar to the certification-based technique, but instead of relying on a deterministic certification test, the delegate checks if the execution of a transaction is serialisable and sends the outcome (abort or commit) of the transaction using a reliable broadcast.

Distributed Locking. This technique fits into the update-everywhere, linear interaction, voting category (Section 3.2.1.D). This is the classical read-one write-all (ROWA) technique. Each operation is broadcast to all replicas where locks are first acquired (hence the name); the operations are then executed. Traditionally, deadlock detection is done using a timeout mechanism, but implementing such a mechanism would introduce a additional parameter in the simulator. Also, as timeouts are not deterministic, this would imply that the locking system would be different from other techniques and make the comparison less valid. To avoid these problems, distributed deadlocks are resolved by building the wait-for graph. The cost of the deadlock detec-

tion mechanism is not included in the simulation, so the distributed locking technique relies on a perfect, cost-less deadlock detector. This is not much of an issue, because distributed deadlocks were very rare in most simulation settings.

Primary-Copy. This technique fits into the primary-copy, constant interactions, non-voting technique (Section 3.2.2.A). This is typical of cold standby primary-copy techniques.

Lazy Update-everywhere. This technique is an implementation of a lazy update-everywhere replication scheme. This implementation has no conflict detection, nor any reconciliation function. This technique simply executes the transaction on the delegate and ships the updates to the other replicas. No consistency checks are done. As the processing is minimal, no other replication technique can outperform this one. The price to pay is that such a system violates the ACID properties. This technique has been implemented in the simulator to give a upper bound for performance, lazy replication is considered much more efficient than eager replication [GHOS96].

5.1.2 Client Module

The clients of the replicated database system were also modelled in addition to servers. Clients are simple sources of transactions. Clients submit one transaction, wait until it is processed, sleep for some time, and start the work cycle again. A client can only submit one transaction at a time — multiple sources are modelled with multiple clients. The most important parameter of clients is the time between submitting transactions. This parameter is the time between the start of two transactions. If a client starts a transaction at time t_1, once the transaction is finished, the clients waits until time $t_1 + d$ before issuing the next transaction. The variable d is a random variable with an exponential distribution so that the mean of d matches the requested interval between transactions. By adding some randomness, we avoid "bursty" load situations.

Typically, a server has many clients attached. If the replication strategy is update everywhere, each server has the same number of clients attached. If the replication technique is primary-copy, only the primary has clients attached. Clients do *not* consume any network bandwidth: we consider that the network interface that interconnects the servers is separate from the network interface used to communicate between clients and servers.[1]

Clients gather all the performance data and compute statistics. The clients also control simulations runs and experiment settings, typically stopping the simulation when the results obtained fit into a certain confidence interval.

[1] Client's request do not consume the CPU resource when being delivered, this is considered part of the transaction parsing and optimising and is not considered here.

5.2 Simulation Settings

We performed an extensive set of simulations to compare the different replication techniques. All techniques shared the same infrastructure layer and the same operational parameters. The main performance metric is the mean response time observed by clients. Simulation where run until this value was at least within a 95% confidence interval with a half width of 5% of the mean response time; often, better confidence interval were obtained. To avoid skewed measures due to initial startup factors [Jai91], the response times associated with the first 500 transactions are discarded.

Parameter	Value
Number of items in the database	10'000
Number of Servers	9
Number of Clients per Server	2 or 4
Disks per Server	2
CPUs per Server	2
Transaction Length	10 – 20 Operations
Probability that an operation is a write	50%
Probability that an operation is a query	50%
Buffer hit ratio	20%
Time for a read	4 - 12 *ms*
Time for a write	4 - 12 *ms*
CPU Time used for an I/O operation	0.4 *ms*
Time for a message on the Network	0.5 or 0.07 *ms*
CPU time to send/receive a message	0.5 or 0.07 *ms*
Time for a broadcast on the Network	0.5 or 0.07 *ms*
CPU time to send/receive a broadcast	0.5 or 0.07 *ms*

Table 5.1: Simulator parameters

In general, simulations were run with the operational parameters in Table 5.1, with one or two parameters being the variables of the experiment. There are two load settings. The first consists of 18 clients connected to 9 servers, either 2 clients per server if the technique is update everywhere, or 18 clients connected to the primary if the technique is primary-copy. The other settings consists in 36 clients connected to 9 servers, with 4 clients per server for update everywhere techniques and 36 clients connected to the primary in the primary-copy case.

The data set contains 10'000 items. Servers are composed of two CPUs and two data disk units. Each CPU has access to any disk, but only one CPU can access a disk at a time. Data items are distributed on the different data disks. The transaction length is uniformly distributed between 10 and 20 operations. Each transaction is either an update transaction (50%) or a query transaction (50%). Queries contain only read operations, updates contain both read (50%) and write (50%) operations.

Read and write operations access one item of database (uniform distribution). A write operation might overwrite a read data, with uniform probability $\frac{1}{database\_size}$.

Operating a read or a write operation uses the disk between 4 and 12 *ms* (uniform distribution). Read operations have a 20% chance of hitting the cache, and therefore occur no disk usage. Each input/output operation (read and write) has a CPU overhead of 0.4 *ms*. The database settings where based on numbers in the literature [ACL87, NSB97, HAA99b, Kem00].

Network settings are based on observed values on a cluster of PC machines. Each machine is equipped with a 733 MHz processor and a 100 Mb/s full duplex network interface. The machines are connected using an Ethernet hub. Network performance was estimated using the Neko framework [UDS01] by sending short messages of approximately 256 bytes. Sending a point-to-point message consumes 0.07 *ms* of CPU at the sender, 0.07 *ms* of the network resource and 0.07 *ms* of CPU at the receiver. We assume a low-level multicast facility (like IP-multicast) with which we can send a multicast in a single operation. The cost is 0.07 *ms* on the network, and 0.07 *ms* of CPU at both the sender and the receiver. We also did experiments using settings that roughly represent a 10 Mb/s network with slower network adapters. In this case, the cost of sending and receiving a message is 0.5 *ms*, and the the cost of message transmission is 0.5 *ms*. We call the first setting (100 Mb/s) the *fast network* setting, and the second the *slow network* setting (10 Mb/s).

5.3 Experiments

We performed several experiments to compare the different database replication strategies. This section presents the different experiments and their results. Each experiment explores different aspects of a replicated database. Section 5.3.1 presents the overall performance of the different techniques, Section 5.3.2 explores the issues of scalability, that is how the techniques behave when the number of server increases. Section 5.3.3 examines how certain replication techniques can distribute the execution of queries. Section 5.3.4 shows how group communication-based replication techniques perform in a WAN setting. Section 5.3.5 presents experiments that compare group-safe replication to 1-safe replication. Section 5.3.6 compares the performance of optimistic active replication with active replication. Finally Section 5.4 discusses the different results.

5.3.1 General Performance

5.3.1.A Description of the Experiment

The first experiment aims at comparing the performance of the different replication techniques under moderate-high load, with a medium number of servers (9). We compared the performance of the different replication techniques by varying the system load. This was done by changing the time interval that clients wait before

issuing a new transaction (this time is measured between two transaction starts). The time between transactions was between 900 *ms* and 1'800 *ms* in 50 *ms* increments. The experiment was done once with two clients and once with four clients per server. With two clients, the load varied between 10 and 20 transactions per second, with four clients, the load varied between 20 and 40 transactions per second. This experiment was done once with the slow network settings (10 Mb/s) and once with the fast network settings (100 Mb/s). This gave four basic settings which are summarised in Table 5.2. The other operational parameters for the experiment are described in Section 5.2.

Medium Load 10 - 20 trx/s	Slow Network 10 Mb/s	Medium Load 10 - 20 trx/s	Fast Network 100 Mb/s
High Load 20 - 40 trx/s	Slow Network 10 Mb/s	High Load 20 - 40 trx/s	Fast Network 100 Mb/s

Table 5.2: Settings for simulating the overall performance of replication techniques

5.3.1.B Results

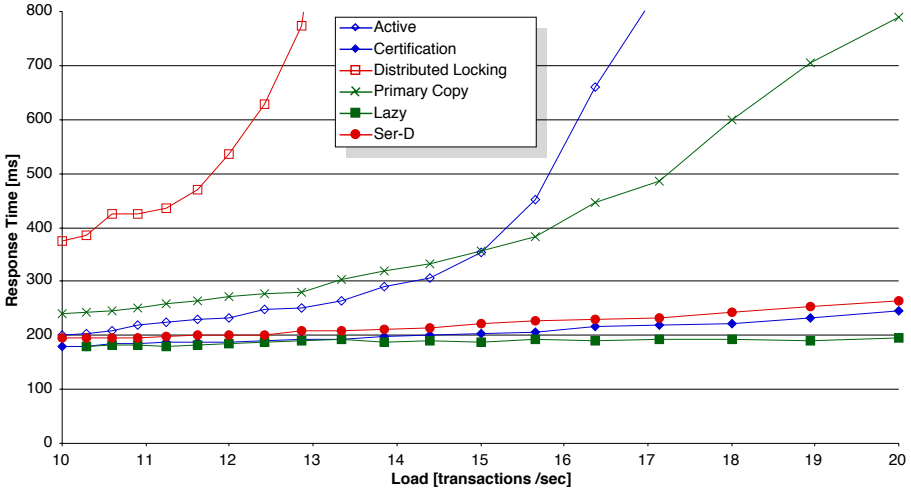

Figure 5.2: Overall performance of replication techniques – medium-load, slow network

Medium Load, Slow Network Figure 5.2 illustrates the result of the experiment with 2 clients per server and the slow network. The experiment shows the performance of different replication schemes under the same operating conditions. The X axis represents the load of the system, expressed in transactions per second. The Y axis represents the average response time of committed transactions. Each replication technique is represented by one performance curve.

The observed conflict rate in the local database managers changed depending on the replication technique: at low load situation, it was below 5%, at high-load it would reach above 20%.[2] The abort rate for all techniques was below 5%, and is not shown.

The lazy replication scheme does not enforce consistency. It is presented to give a reference in the form of the best performance that can be achieved in this setting. Because this technique does little extra processing and does no synchronisation at all, its performance is very good and is not affected by changes in the load. Basically, the performance of lazy replication with n servers is equivalent to n non replicated servers getting $1/n$ of the load.

In low load situations (left part of the graph), most techniques have very similar performance: certification, Ser-D, active and primary-copy have basically the same response time. This reflects the fact that all those techniques fit in the same category, with a constant number of network interactions. Lazy replication does outperform those techniques by a rather small margin. The main advantage of lazy replication is very good load balancing, but as the load is limited, this results in a small difference. This explains why there is no noticeable difference between techniques that do load balancing (certification-based replication), and others that do not (primary-copy and active replication).

Distributed locking has a response time that is 60% higher. This is caused by contention on the network, as distributed locking sends one broadcast per operation. This leads to a situation where the network becomes the bottleneck – in the simulator, the usage rate of the network resource quickly reached 100%.

As the load increases, we can see that the response time of all techniques increases, yet the relative performance of the different techniques changes. The response time of distributed locking increases and shows that there is an asymptotic limit to throughput around 13 transactions per second. This is caused by high network contention. A similar phenomenon is observable with active replication, which has a maximum throughput around 16.5 transactions per second. The reason, this time, is not the network (at the highest load, the network usage rate was around 6%), but high load and the important serialisation phase needed to enforce determinism. Transactions need to be serialised until their locks are granted and active replication does no load balancing: all transactions are executing on all replicas. Those two factors form the bottleneck of active replication: high load causes lock contention, and lock contention slows down the serialisation phase. This serialisation phase becomes a bottleneck. Primary-copy replication also has a load problem, as most of the work is done on the same server, so performance tends to degrade as the load increases.

The performance of the Ser-D technique and the certification-based technique remains very close to lazy replication. It is interesting to note that the response time of the Ser-D replication is around 15 *ms* higher than the response time of

[2] The observed conflict rate was calculated by marking transactions that had to wait on a lock held by another transaction during their execution.

the certification-based technique. This is explained by the differences between both techniques. The Ser-D techniques is weak voting, so before a transaction can commit, all replicas must wait for the delegate to decide the outcome of the transaction. The certification based technique has no such need, all replicas decide on the outcome of the transaction using the total order of delivery. Still, the difference between certification and Ser-D techniques cannot be explained only by the cost of a broadcast which "costs" around 1.5 *ms*, the 15 *ms* difference is mostly explained by the cost of coordination: all servers need to wait for the delegate to finish processing t in order to be able to terminate t locally. In other words, the time needed to broadcast the data is negligible, but the time lost waiting is not.

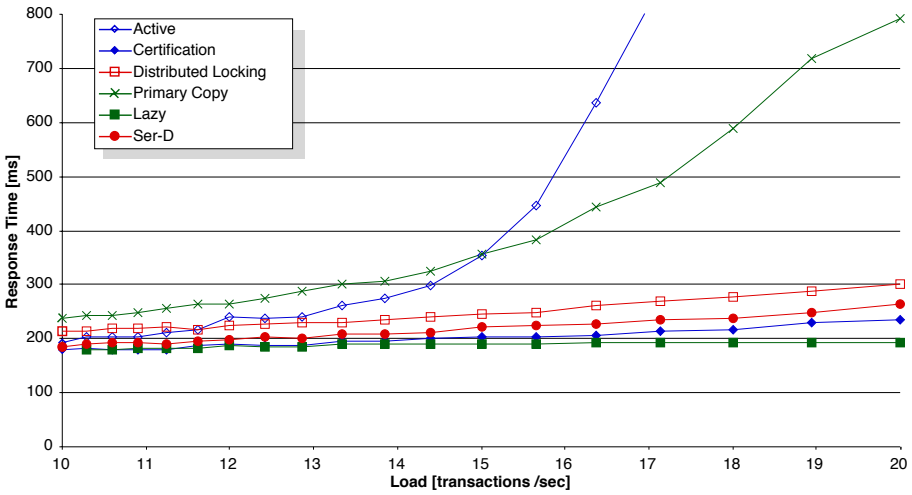

Figure 5.3: Overall performance of replication techniques – medium-load, fast network

Medium Load, Fast Network Figure 5.3 shows the results of the same experiment but with a fast network (100 Mb/s). While most techniques behave in a very similar fashion, one technique has a very different performance curve: the distributed-locking technique. This makes sense: as the network was the bottleneck of this technique in the previous experiment, a faster network implies a different performance. While performance of distributed locking is much better with a fast network, the response time is still higher than the response time of group communication-based techniques ($\approx$ 50 *ms*). This difference can only be partly explained by network usage (it takes around 0.2 *ms* to send a message), with an average of 15 operations per transaction and two messages per operations (request lock and confirm), this gives a network overhead of around 6 *ms*. This means that the major part of the performance penalty is related to the way transactions are processed. This processing overhead is probably also partially responsible for the overhead of the distributed-locking technique with the slow network (previous

experiment), but the majority of the overhead with the slow network was due to network contention.

Another interesting thing to note when comparing Figures 5.3 and 5.2 is that the difference between the certification-based technique and the Ser-D technique stays roughly the same. This shows that the difference between those two techniques is not related to the use of the network, but the way those two techniques are structured.

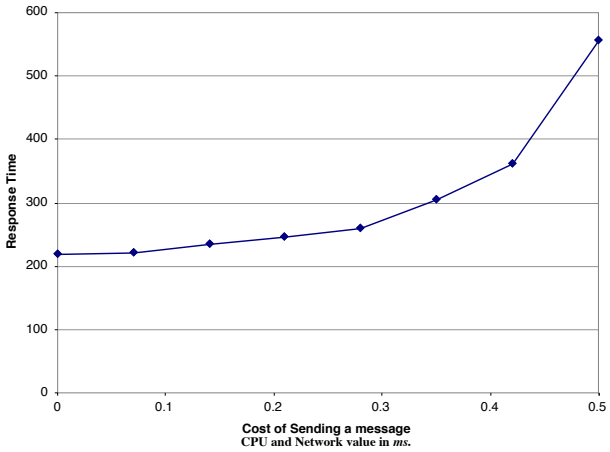

Figure 5.4: Influence of network performance on the distributed-locking technique

Network Cost and Distributed Locking To understand the relationship between the performance of the network and the response time of the distributed locking technique we measured the response time of the technique while changing the performance of the network. The result of this experiment is illustrated in Figure 5.4. In this experiment, we changed the "cost" of sending a message on the network and plotted the response time of the distributed-locking technique with a interval between transactions of 1500 *ms*, resulting in a load of 12 transactions per second. The X axis represents the cost of sending one message – this cost is changed for both CPU and network. When $X = 0.07$ *ms* this corresponds to the settings of the fast network, when $X = 0.5$ *ms* this represents the slow network. The Y axis represents the response time in milliseconds. The graph shows that the response time of the distributed-locking technique increases with the cost of networking operations. We also see that this curve is not linear: When the network becomes the bottleneck, the response time increases much more: this can be seen when the cost of the network is at 0.35 *ms*. At this point, the network facility is used at 67% and on average, processes had to wait more than 2 *ms* to access the network resource.

High Load, Fast and Slow Network Figure 5.5 shows the result of the experiment with 4 clients and the slow network. Only the curves of the Ser-D, certifica-

tion and lazy techniques were plotted, the others being unable to sustain more than 20 transactions per second. Figure 5.6 shows the results of the experiment with 4 clients and the fast network.

Figures 5.5 and 5.6 are very similar, except for the fact that the distributed-locking technique is not present in the slow network case (in this setting distributed-locking cannot sustain such high loads). As the main difference between both experiments is the behaviour of the distributed-locking technique, we will concentrate the discussion on the experiment with the fast network, so as to include distributed locking. Except for distributed-locking, all considerations are the same, as the influence of network performance on overall performance is weak.

With both the fast and the slow network, we see that performance degrades steadily when the load is around 30 transactions per second. At first glance, the behaviour of the certification technique is much better that distributed locking and Ser-D: the response time stays significantly lower even when the system starts to become overloaded. In fact, in the fast network setting, when the load reaches 32 transactions per second, the response time decreases. When the load is above 34 transactions per second, the response time of the certification technique is within the confidence interval of lazy replication.

The difference lies in the abort rate of the different techniques. While in moderate load situations, the abort rate of the different techniques were marginal, in high-load situation, the abort rate becomes significant. Figure 5.7 shows the abort rate of certification, Ser-D and distributed locking techniques in high-load situations (fast network) – the parameters are the same as in Figure 5.6. We see that while the response time of certification-based replication is low, its abort rate is significantly higher. The overload of the system yields a high conflict rate in the certification phase, therefore a lot of update transactions abort. In fact most of the aborts are update transactions. We can also see that distributed locking has a sharp increase of aborts once the load reaches 28 transactions per second: at this point deadlocks start to become significant. It is interesting to note that while the response time of Ser-D replication increases, the abort rate stays stable, below 2%.

5.3.1.C Discussion

The performance measurements show that replication techniques can be split into four categories: network bound replication techniques (distributed-locking), performance bound techniques (active and primary-copy), efficient group communication-based techniques (Ser-D and certification) and the lazy technique.

Distributed-locking is affected by network performance, most other techniques are not: their performance is similar with both the slow and the fast network. This is due to the fact that they rely on a single broadcast operation, either a simple broadcast (primary-copy and lazy) or a total order broadcast (group communication-based techniques).

The performance of group communication-based techniques depends mostly on their architecture: techniques that where designed for database replication (cer-

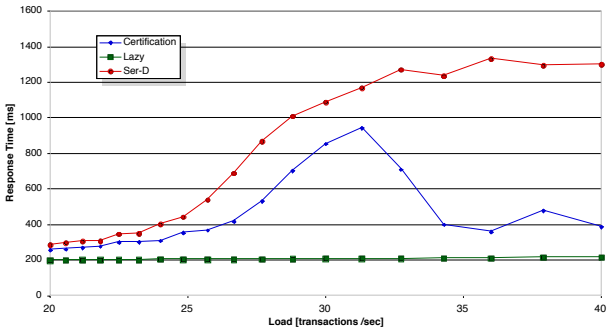

Figure 5.5: Overall performance of replication techniques – high-load, slow network

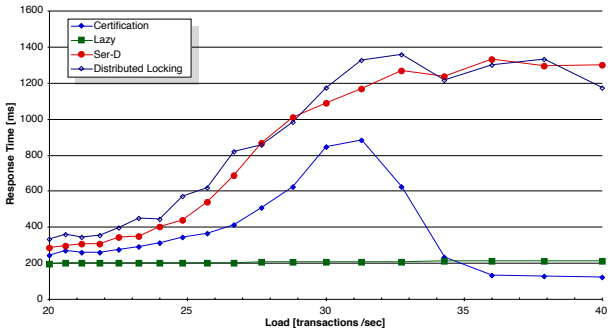

Figure 5.6: Overall performance of replication Techniques – high-load, fast network

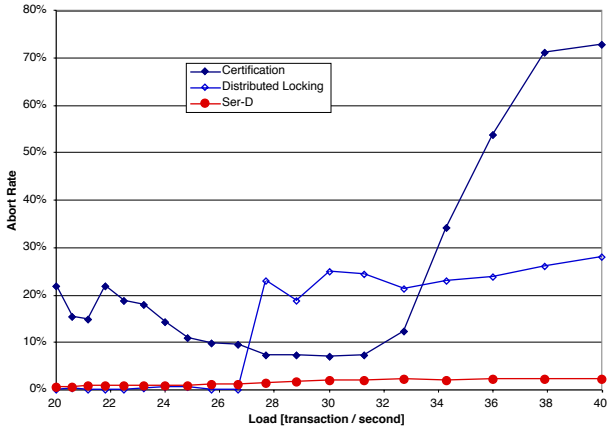

Figure 5.7: Abort rate in high-load situation (fast network)

tification and Ser-D) outperform significantly the basic technique (active replication). The culprits for active replication's bad performance are the serialisation phase and the lack of load balancing.

Even in a fast network configuration, distributed-locking is outperformed by efficient group communication-based techniques. The reason for this is that the execution of transactions in distributed-locking replication is tightly coupled between the replicas. All those synchronisation phases cause a serious slow-down. Synchronisation is also the cause of the performance difference between Ser-D and certification-based replication. The Ser-D technique has one synchronisation phase (weak voting phase), that "costs" this technique around 50 *ms* when compared to the certification technique.

The behaviour of certification-based replication in high-load situations could be alleviated with flow-control techniques. Flow-control would help to avoid situations where the conflict rate causes too many aborts and thus the technique becomes unusable.

5.3.2 Scalability

5.3.2.A Description of the Experiment

One important aspect of replication techniques that is worth analysing is their scalability. A good replication technique must be usable even if the number of replicas is high. In this experiment we measured how the system reacted to a changing number of servers. The interval between transactions was fixed at 1800 *ms* and the number of clients at 36, thus resulting in a load of 20 transactions per second. We then changed the number of servers and observed the performance of the system. The different configurations are listed in Table 5.3. As primary-copy only requires one broadcast, this technique is very scalable. Therefore we concentrated on the scalability of update everywhere techniques. We did the experiment with the fast network and two different types of transaction loads. The first is called "mixed load", and contains 50% of queries. The second is called "mostly queries" and contains 80% of queries.

Servers	Clients per Server
36	1
18	2
12	3
9	4
6	6
4	9
3	12
2	18

Table 5.3: Scalability configurations

5.3.2.B Results

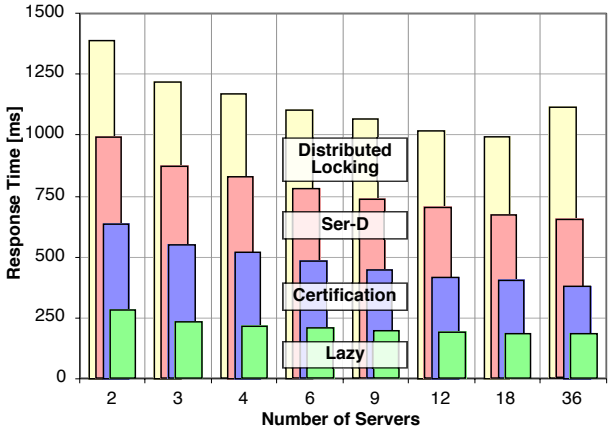

Figure 5.8: Scalability of update everywhere techniques with a 50% query load

Mixed Load Figure 5.8 shows the results of the experiment with a query load of 50%. We compared the response time of the following replication techniques: certification-based replication, Ser-D, distributed-locking and lazy. Each set of bar represents the performance with a given number of servers, starting with 2 servers for the leftmost bar. Each technique is represented with a different bar, the height of each bar (Y axis) represents the response time in milliseconds.

The general performance is similar to the experiment presented in Section 5.3.1: lazy replication outperforms all other techniques. Group communication-based techniques outperform distributed-locking. In general, we see that the response time decreases as the number of replica increases. This shows that as the number of servers grows, some part of the load (queries) can be distributed on more replicas, thus giving better performance. The most interesting part appears in the extreme case, when the number of replicas is maximal (36 servers): the performance of distributed-locking decreases significantly. This is not due to network usage (which stays below 5%), neither to distributed deadlocks (the abort rate was below 1%), but to the cost of coordination that is inherent to this technique.

While aborts had no significant impact on distributed-locking, this was not the case for the certification technique. Figure 5.9 shows the abort rates for both the certification technique and the Ser-D technique. The X axis represents the number of servers, the left Y axis represents the abort rate, in percentage. We see that while the abort rate of the Ser-D technique is stable below 2%, the abort rate of the certification technique increases with the number of servers. When the number of server is maximal, the abort rate reaches 20%!

A first hypothesis to explain this behaviour is related to the difference on how local and global conflicts are handled. If two transactions conflict, the conflict can

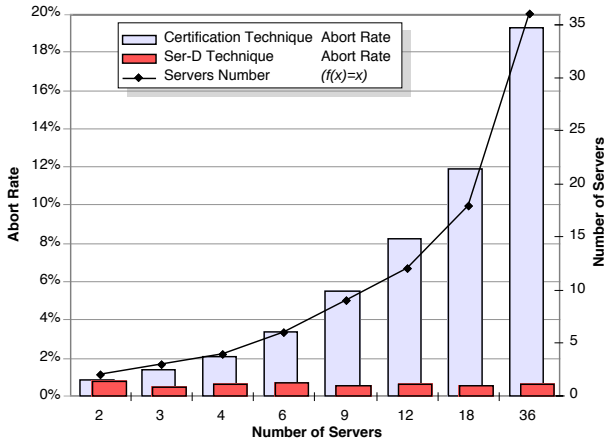

Figure 5.9: Correlation between conflict function and observed aborts

be handled in two ways. If both transactions originate from the same server (they have the same delegate server), the conflict will be handled by the local locking mechanism. The local locking system will serialise the execution of the conflicting transactions: if no deadlock occurs, there will be no abort. If both transactions originate from different servers, the conflict will only be detected at certification time, and will result in the abort of one transaction. So as the number of replicas increases, so does the chances of abort. So if two transactions t_a and t_b conflict, the chances that the conflict results in an abort is proportional to the probability that they are on different servers: $1 - (1/n)$, where n is the number of servers.

Still, the observed abort rate cannot be explained by this phenomenon. First, the observed conflict rate is roughly proportional to n: the black line on Figure 5.9 represents the linear function $f(x) = x$ (the right Y axis represents the number of servers) and is clearly correlated with the abort rate. Secondly, the conflict rate is much too low to explain such an abort rate. This can be seen by comparing the abort rate of both the certification and the Ser-D techniques. The Ser-D technique is very similar to the certification-based technique: both execute transactions in the order of delivery by the total order broadcast and check for conflicts. The certification technique uses a deterministic check mechanism, while the Ser-D relies on the delegate server to detected actual conflicts. The certification needs to "guess" what transaction might conflict, and might therefore do unnecessary aborts. So while the number of aborts of the certification technique might be slightly higher than the abort rate of the Ser-D technique, both abort rates should be roughly the same. Figure 5.9 also shows the abort rate of the Ser-D technique: clearly, this rates stays low. So *real* conflicts cannot explain the abort rate.

The actual issue is related to the way the certification algorithm was implemented. When a transaction is delivered, it is placed in a conflict list. This list is used to check for potential conflicts by the certification test. When a transaction

has been committed on a replica, it becomes *stable* on this replica. The information that a transaction is stable is piggy-backed on subsequent total order broadcast messages. Once a transaction is known to be stable on all replicas, it is removed from the conflict list. So a transaction is removed from the conflict list once it has committed on all replicas, **and** all those replicas have sent a total order broadcast message.

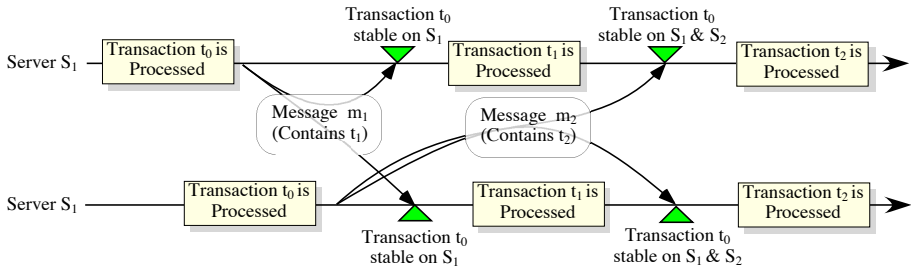

Figure 5.10: Conflict list scenario

This means that even if transactions are executed sequentially (so there should be no conflict), conflicts appear because transactions stay too long on the conflict list. Figure 5.10 illustrates this problem in the case of a system with two servers. The system consists of two replicas: s_1 and s_2. Initially transaction t_0 is in the conflict list. Transaction t_0 will be removed from the conflict list on all replicas once two total order broadcast messages have been delivered: m_1 (from s_1) and m_2 (from s_2). Each message contains one transaction, that we call t_1 and t_2 respectively. Let us assume that m_1 is delivered before m_2: when m_1 is delivered, all replicas know that t_0 is stable on s_1 (the sender of m_1), but the status of t_0 on s_2 is unknown, so t_0 stays in the conflict list. All replicas then start to process the content of m_1, transaction t_1. As t_0 is still in the conflict list, t_0 can cause the abort of t_1 if they conflict (even though t_0 might be terminated on all replicas). When m_2 is delivered, the system knows that t_0 is now stable on s_2 (the sender of m_2). As t_0 is stable on all replicas (s_1 and s_2) it can be removed from the conflict list. Therefore t_0 can cause the abort of t_1, but not t_2.

Now if we have n replicas, transaction t will be removed from the conflict list once messages $m_1 \dots m_n$ have been delivered. This means that t_0 can potentially conflict with transactions $t_1 \dots t_{n-1}$. So the number of potential conflict is proportional to the number of servers n, **regardless** of the load of the system.

To verify this hypothesis, we measured the abort rate of the certification technique with varying loads (between 1 and 30 transactions per second) and a large number of servers (36). Figure 5.11 shows the results of this experiment. The X axis represents the load of the system, the Y axis the abort rate. We see that while the load varies by a factor of 30, the abort rate stays the same, slightly below 20%.

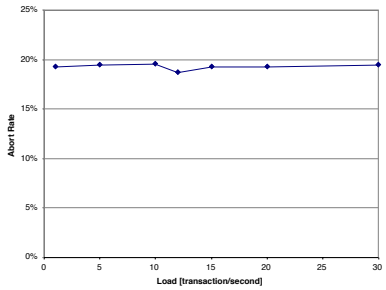

Figure 5.11: Abort rate with 36 servers with changing loads

"Mostly Queries" Load Figure 5.12 shows the experiment with a query load of 80%. The overall graph is similar to the one in Figure 5.8, with better response time – this is expected as there are more queries in the workload. We also see that while the increase is not as significant as with 50% of queries, the performance of distributed locking still starts to degrade when the number of servers is high (36).

5.3.2.C Discussion

In general, group communication-based replication techniques scale well assuming a moderate load and a large proportion of queries. The response time does not increase with the number of servers, but diminishes, as one would expect. We showed that the certification technique has a problem that leads to many aborts when the number of servers is high. The Ser-D technique has no such problems. The abort problem of the certification technique can be addressed in two ways:

- Using a version-based database. In this case the certification test would then

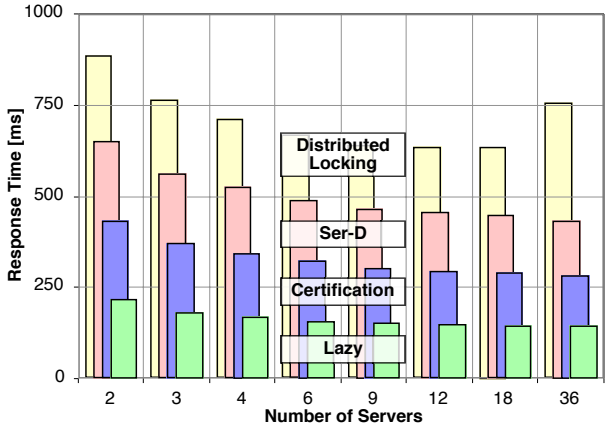

Figure 5.12: Scalability of update everywhere techniques with a 80% query load

rely on the version number of items instead of a conflict list. While this approach solves the problem, its drawback is that it imposes some requirements (access to version numbers) on the database system.

- Use additional messages. When a transaction t is stabilised on server s_i, this server sends a total order broadcast message[3] that contains no transaction but signals that t is stable on the sending replica. This message is sent after t committed, and does not impact directly on the response time of t's processing. The main problem with this approach is that it generates a message explosion. If there are n servers, then the processing of each transaction will require $n + 1$ total order broadcast messages (1 for the algorithm and n for distributing the information that t is stable. Even with an efficient total order broadcast algorithm, the impact of all those messages would be important.

The interesting aspect of group communication-based replication is that issues like scalability are resolved outside the replication scheme: if the total order broadcast primitive scales well [RFV96], then the replication technique will scale well.

5.3.3 Query Proportion

5.3.3.A Description of the Experiment

The proportion of queries in the load of the system can have an important effect on the performance of a replicated database. If the read-write policy of a replication technique is ROWA, queries only need to be executed on one replica. In some cases, they can be executed without requiring any communication. By changing the query rate, we can compare the load-balancing capacities of each replication strategy.

To measure how different replication techniques handle queries we fixed the load of the system, changed the proportion of queries submitted in the load, and measured changes in the response time. When the query proportion is 100% there are no update transactions; if the query proportion is 0%, all transactions are updates. The query proportion was increased from 0% to 100% in 10% increments. We measured the impact of the query load in two settings:

Low-Load In this setting, the system is configured with 36 clients and 4 servers connected by the *fast* network. The interval between transactions is fixed at 3'600 *ms*. This yielded a load of 10 transactions per second. When the query rate is 50%, this setting corresponds to the left edge of Figure 5.3.

Moderate-Load In this setting, the system also consists of 36 clients and 4 servers connected by the *fast* network. But this time, the interval between transactions is 1'800 *ms* and thus a load of 20 transactions per second. When the query rate is 50%, this setting corresponds to the right edge of Figure 5.3.

[3] The message must be a total order broadcast, because it updates the state of the certification test module, which must behave in a deterministic way – the same for all servers.

5.3.3.B Results

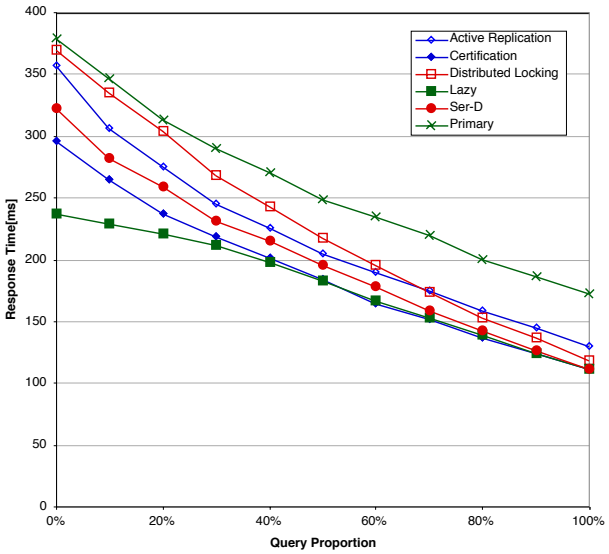

Figure 5.13: Performance with changing query rate at low-load (10 transactions per second)

Low Load Figure 5.13 shows the results of the experiment at low-load (10 transactions per second). When the query rate is 100%, the load consists of only read operations and transactions can be distributed on all servers. In this situation, certification and Ser-D replication have the same performance than lazy replication: the load is perfectly balanced between all servers and no communication occurs. Distributed-locking suffers from a slight overhead because of the protocol complexity, but has roughly the same performance.

The two remaining techniques, active replication and primary-copy have worse performances because they do no load balancing. In both techniques, all the work is done on one server. In the primary-copy case, all the work is done on the primary. In active replication, the situation is slightly different: all the work is done on *all* replicas – so technically, all replicas do the work of the primary. In both cases, there is no load-balancing.

As both techniques handle the load in the same way, one would expect that they have the same performance. In fact primary-copy should outperform active replication by the cost of a total order broadcast. Yet the results show that active replication outperforms primary-copy replication by approximately 30%. The reason for this lies in the way active replication works. In active replication, the delegate server merely acts as a proxy for all servers: when a transaction is delivered, it is sent to all servers (including itself) using a total order broadcast. Each server processes the transactions and sends the results back to the delegate, the delegate forwards the *first* response to the client. So in practice, this means that

the perceived response time of the client is the response time of the fastest server. So the observed response time r is the minimum response time for all replicas: $r = \min\{r_1 \ldots, r_n\}$.

Why are there difference between the response times of servers? The response time of a transaction on one replica depends on multiple factors: actual load of the system, number of items in cache and time needed for a seek. These factors are, to some extent, random[4]. So as the number of replicas increases, the observed response time will improve more and more, as we are more likely to get a fast first response. Here we benefit from the fact that all replicas do all the work, i.e., that the technique does no load balancing. This improvement is similar to the read operation improvement in RAID level 1 systems [CT92].

As the proportion of queries diminishes, the performance of all techniques degrades. This performance degradation is more noticeable for distributed-locking: as the number of writes increases, the overhead of this technique becomes more obvious. It is interesting to note that the performance of the certification-based technique cannot be distinguished from the performance of lazy replication if the query proportion is larger than 40%. This shows that for high query rates and low load, certification-based replication is close to optimum.

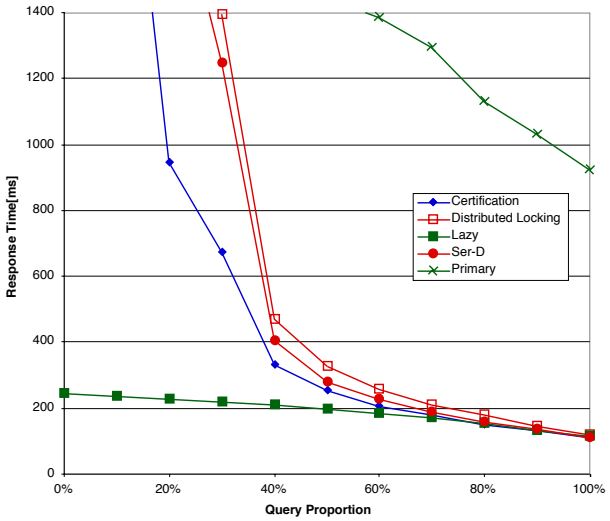

Figure 5.14: Performance with changing query rate at moderate load (20 transactions / second)

[4]Caching algorithms that are usually used are not random, but rather deterministic. Classic deterministic caching algorithms would not benefit active replication, because all the replicas would have the same caching behaviour. In the simulation, caching was simulated as a statistical process, so it benefited active replication. Special algorithms could be tailored for active replication and give even better benefits: for instance by having each replica keep in cache only a subset of the data.

Moderate Load Figure 5.14 shows the results for the same experiment with a higher load (20 transactions per second). The first thing we see is that some techniques have very bad performance: active replication (whose curve is outside the graph), and primary-copy replication (upper right corner). Most other techniques have non-linear response curves: if the query proportion is to low, performance drops suddenly.

If we compare this graph with the one in Figure 5.13 it is interesting to see that the relative performance between active replication and primary-copy replication is inverted: while both techniques perform poorly, primary-copy has better performance. In high-load situations, the advantages of selecting the fastest response is offset by the fact that much more processing is needed. Additionally, the serialisation phase of active replication becomes a bottleneck in high-load situations (see Section 5.3.1).

Changing load and query rate To understand how the linear curves in Figure 5.13 transform into those in Figure 5.14, we plotted 3 dimensional graphs for the following techniques:

- Certification (Figure 5.15)
- Ser-D (Figure 5.16)
- Distributed-locking (Figure 5.17)
- Primary-copy (Figure 5.18)

We did not plot the remaining two techniques, as their performance in moderate load setting is very bad. For each technique, the X axis represents the query proportion, the Y axis (depth) represents the load of the system in transactions per second and the Z axis (height) represents the resulting response time. Each technique is represented by a separate surface.On each surface, we marked the limit where $Z = 200$ *ms* and where $Z = 300$ *ms*.

First we see that all four surfaces are continuous: this shows that all four techniques are stable in the parameter space considered. The most noticeable aspect of those four figures is that distributed-locking, certification and Ser-D replication have the same general shape: performance is good if the load is low or the query proportion high. Response time reaches a “peak” in high-load, low query proportion situations. Primary-copy (Figure 5.18) on the other hand, has a very different general shape. While the technique is also sensitive to some extent to the query proportion, the load has a far greater influence on performance. As the load increases, the response time forms a “wall”, even if the load is composed only of queries. The reason for this is that primary-copy does no load-balancing.

Let us consider the parameter values where the response time reaches 200 *ms* and 300 *ms* respectively, we can see the performance difference between the distributed-locking technique on one hand (Figure 5.17), and the group

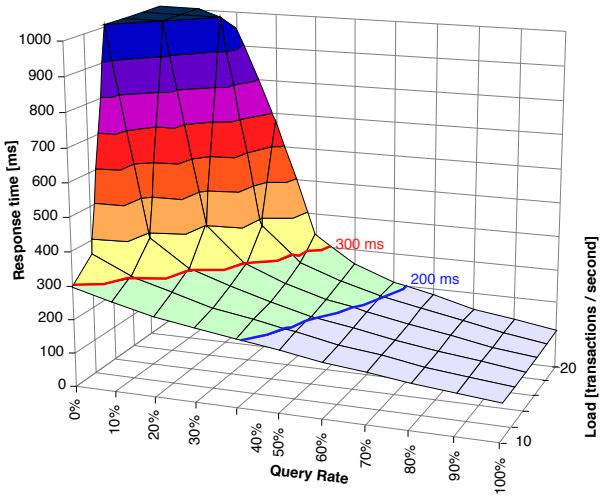

Figure 5.15: Performance of certification-based Replication with changing query proportion and changing loads

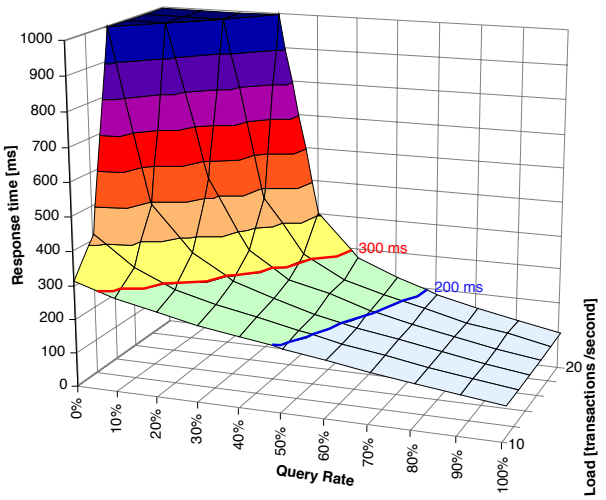

Figure 5.16: Performance of Ser-D replication with changing query proportion and changing loads

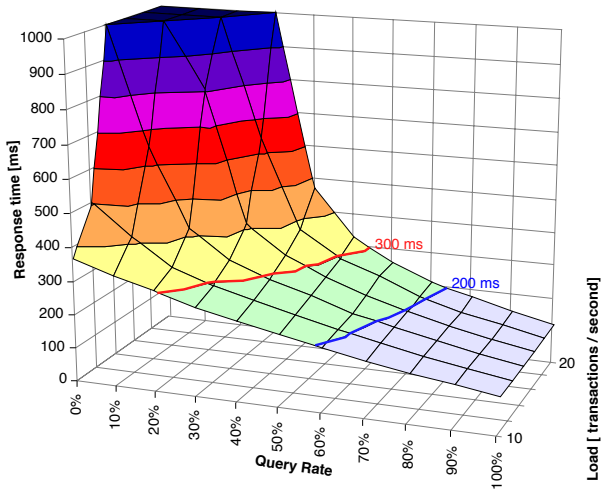

Figure 5.17: Performance of distributed-locking replication with changing query proportion and changing loads

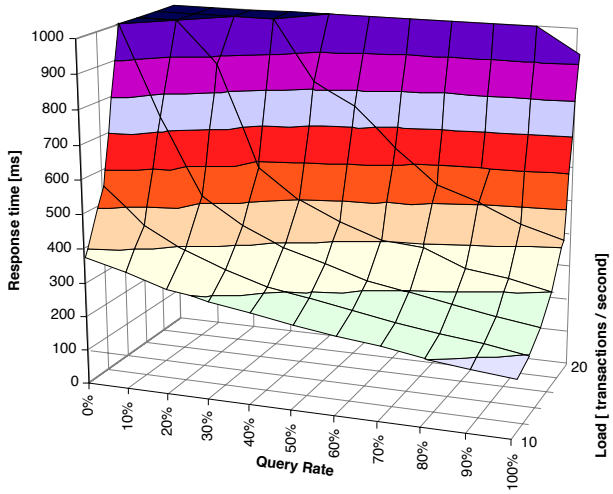

Figure 5.18: Performance of primary-copy replication with changing query proportion and changing loads

communication-based replication techniques (Ser-D and certification, Figures 5.16 and 5.15) on the other hand. Group communication-based replication techniques outperform distributed-locking systematically. In general the difference is roughly equivalent to a difference of 10% of the query proportion, e.g. distributed-locking behaves with a 40% query proportion like group communication-based replication with a 50% query proportion. This is consistent with the performance difference observed in overall load performance (Section 5.3.1).

5.3.3.C Discussion

The experiment shows that the load balancing features of the different replication techniques has an important impact on performance, especially when the load contains a large proportion of queries. We see that group communication based replication techniques offers good load-balancing. As those techniques are built on optimistic hypotheses (execute first, check for network conflicts after), they work best when the proportion of queries is high. In good conditions (moderate load and high query proportion), those techniques have a performance close to the performance of lazy replication.

5.3.4 Wide Area Network

5.3.4.A Description

All previous experiments were based on local area network (LAN) settings, with servers connected with a moderate (10 Mb/s) or fast network (100 Mb/s). Classical database replication strategies simply cannot cope with the long delays associated with a wide area network (WAN). Yet, we have seen in Section 5.3.1 that group communication-based replication is not very sensitive to the performance of the network – the latency of the network will affect the response time and the time locks are held (write locks are kept until the total order broadcast is delivered), but as there is only one network interaction this influence is limited. Other group communication based WAN database replication scheme have been proposed, but they are based on the primary-copy scheme [FP01]. For this reason, it is interesting to evaluate the performance of Ser-D and certification replication in a WAN context.

To do this, we changed the time for message transmission to 100 *ms*, both for point-to-point an multicast messages. This represents the time for an intercontinental message and is quite large (A ping round-trip from EPFL to the USA is between 100 and 200 *ms* depending on the location).

Contrary to a LAN situation, network contention is not an issue in a WAN context, so it was not modelled. Instead of “using” a resource, the sending of message is simply a wait in the simulator. The system was set-up with 36 clients connected to 9 servers. By changing the interval between transactions, a load between 10 and 20 transactions per second was submitted to the system. The load was composed only of update transactions. Query execution does not involve any network operation and so the performance of query processing is not affected by the long

network delay modelled here; query response time is therefore the same than in a LAN setting. All other parameters are those described in Table 5.1.

5.3.4.B Results

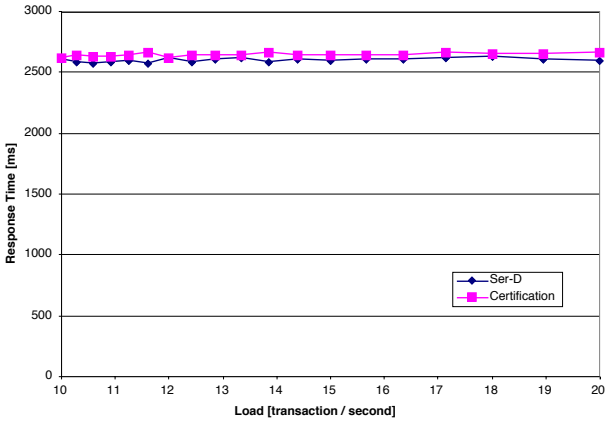

Figure 5.19: Performance in a WAN setting

Figure 5.19 shows the result of this experiment. While the response time is quite high (above 2500 *ms*), it is very stable. The abort rate was also stable, around 4.5%. It is interesting to note that while certification replication outperformed Ser-D replication in LAN settings, the difference in the WAN setting is within the confidence interval.

5.3.4.C Discussion

While large, the response time of group communication-based replication techniques stays reasonable in a WAN setting. Overall, the performance of both techniques is only affected by the the response time of the total order broadcast algorithm. In our case, the total order broadcast algorithm was not one optimised for latency [Déf00], so choosing a different algorithm might improve latency significantly.

5.3.5 Group-Safe replication

5.3.5.A Description

The goal of this experiment was to measure the performance improvement that can be gained by transforming a group communication based replication strategy so that in enforces group-safety (as opposed to group-1-safety). This improvement is discussed in Section 4.4.

For this, we compared three replication strategies: certification, lazy and a group-safe version of certification. The experimental settings are the same than those for the experiment of Section 5.3.1: 36 clients connected to 9 servers generating a moderate load (20-40 transactions per second) and using the fast network (100 Mb/s).

5.3.5.B Results

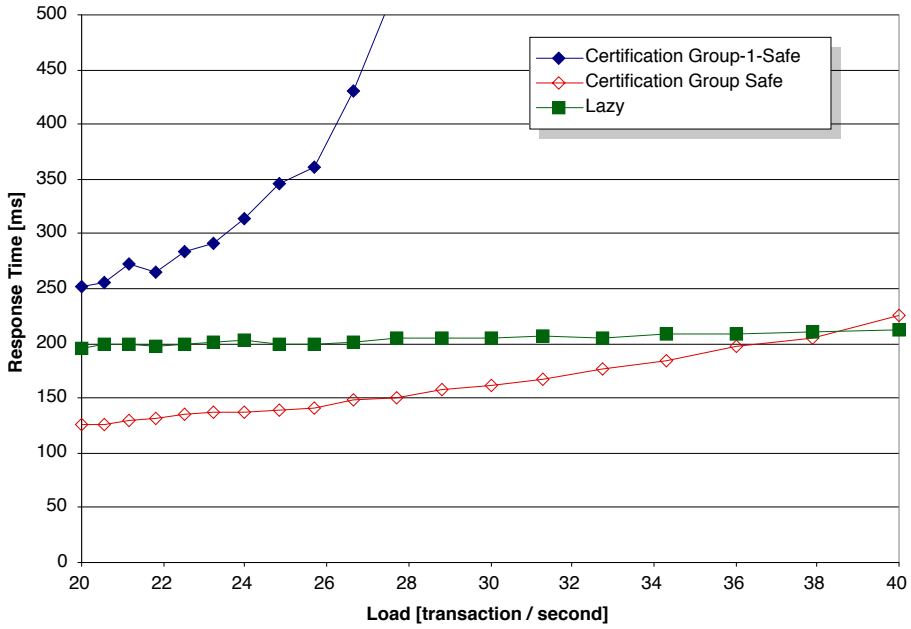

Figure 5.20: Performance of group-safe certification

Figure 5.20 shows the results of this experiment. The X axis represents the load of the system in transactions per second, the Y axis the measured response time, in milliseconds. Each technique is represented by one performance curve. The curve of lazy and certification are the same as in Figure 5.6 for clarity; we removed the part of the certification curve above 30 transactions per second, where the response time drops when the technique aborts most of the update transactions.

The performance curves shows that group-safe replication has very good performance. We see that the group-safe certification technique outperforms even lazy replication when the load is below 38 transactions per second. The abort rate of this technique was constant, slightly below 7%. The very good performance of the group-safe technique is due to the use of asynchronous writes. This means that writes to disk are basically done in a separate thread, outside the scope of the transaction. In high-load situations, group-safe replication loses its advantage over lazy replication.

5.3.5.C Discussion

Group-safe replication offers, as expected, very good performance, and in moderate load situations, outperforms even lazy replication. This shows that transferring the responsibility of durability from stable storage to the group (see Section 4.4) makes sense in a LAN: in our setting, writing a value to disk takes around 10 *ms*, while performing a total order broadcast (which is considered a very complex and costly protocol) takes approximately 1 *ms*.

5.3.6 Optimistic Active Replication

5.3.6.A Description

The previous experiments showed that active replication was not very good when compared to other group communication-based techniques like certification or Ser-D replication. Additionally, active replication requires static transactions, which restricts its usage. The main advantage of active replication is that it does not abort any transactions: transactions are broadcast at the beginning and do not need to pass a certification phase (like certification-based replication) or to wait for a commit message from the delegate site (like Ser-D replication). Static transactions can be executed without any risk of deadlock and are therefore never aborted. All those characteristics make active replication very interesting for real-time applications, where aborted transactions mean unpredictable delays.

In this section evaluate an optimisation of active replication, that we call optimistic active replication. This replication technique was proposed in [KPAS99] under the name Fine Granularity Locking. This technique relies on a early optimistic delivery of messages to start processing earlier. Optimistic early delivery is possible in a LAN setting because of the spontaneous order property (see Section 2.4.6): in a LAN, messages are in the right order most of the time. By starting the processing earlier, the execution of transaction t and the execution of the total order broadcast algorithm that will delivers t's order overlap. If the message ordering generated by the spontaneous ordering does not match the definitive delivery order and a conflict occurs, t will be aborted and restarted. The client does not need to be notified of t's abort and subsequent restart.

Both active replication and optimistic active replication collect the responses of all servers and transmit the first response to the client. This yields improved response time as the response time observed by the client is the response time of fastest server. This gives increased performance for active replication (see Section 5.3.3.B), but also for optimistic active replication – if a transaction t is restarted on a server s_i, then the client will get the response of a server s_j where t did not restart. We call this technique "response collection": the delegate collects the response from all servers (including itself) and forwards the first to the client. All experiments in this section are done using response collection. The same experiments were conducted without response collections, that is servers do not send back their results to the delegate server. In this setting, when a client gets a re-

sponse, it contains the results of the delegate server. In this case, the results were similar but with a higher response times (around 30 *ms* more).

For the optimistic technique to make sense, we need two things: first that the spontaneous ordering property holds most of the time, second that the processing time of a transactions is of the same order of magnitude than the time needed to execute the total order broadcast algorithm. The improvement of the response time will be a portion of the time needed to do the total order broadcast. If processing a transaction is much longer than processing the total order broadcast, the improvement will be negligible. If processing time of a transaction is shorter than the transmission time, then transactions will need to wait for the final delivery in order to commit. As write locks are kept until commit, this will increase lock-contention.

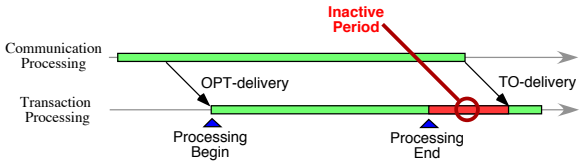

Figure 5.21: Overlap of communication and processing in optimistic active replication

Figure 5.21 illustrates the problem when the processing time of the transaction is shorter than the processing time of the total order broadcast algorithm. OPT-delivery represents the optimistic delivery of the total order broadcast, TO-delivery the effective delivery in total order. The "inactive period" time of transaction processing represents the period of time during which locks are held and no processing occurs.

The experiments for optimistic active replication were done using the parameters described in Table 5.1, with a slow network and 36 clients attached to 9 servers. To represent faster processing, the transaction length was shortened (uniform distribution between 5 and 10 operations) and the cache hit ratio was raised to 80%. As the difference between both techniques are slim, experiments were run until the relative error rate of the response time was below 2%. The time between the start of two consecutive transactions was in the interval from 900 to 1'800 *ms*, which resulted in a load between 20 and 40 transactions per second.

5.3.6.B Results

Overall Performance. Figure 5.22 shows the compared performance of active replication and the optimistic version of active replication. The X axis represents the load of the system, expressed in transactions per second, while the Y axis represents the response time expressed in milliseconds. We measured the performance of active replication and two configurations of the optimistic active replication: one with no out of order messages, and therefore no transactions restarts (no restart),

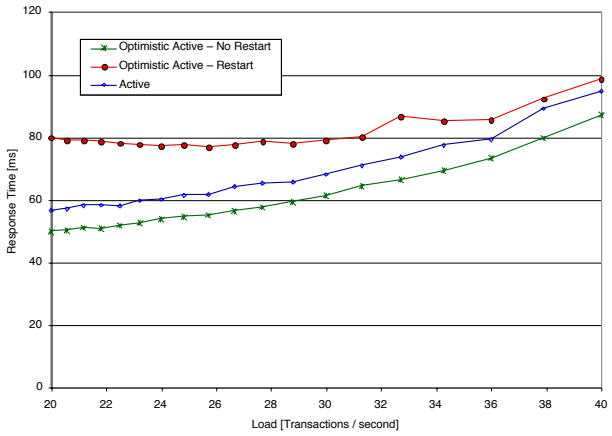

Figure 5.22: Performance of active replication and optimistic active replication

and one with message reordering (restart).

Message reordering is measured by two factors: the proportion of messages that are reordered, and the reordering level. The reordering level expresses by how many positions in the serial order a message can be displaced. For instance if the the correct sequence of messages is $m_1, m_2, m_3, m_4, \ldots$ and the reordering level is 2, when m_1 is delivered out of order, the sequence will be $m_2, m_3, m_1, m_4, m_5, \ldots$ As observed in Section 2.4.6, the message reordering level in a LAN is at worst of 2. In our simulation the reordering level was either 1 or 2 (uniform distribution). Reordering was simulated by tossing a dice for each message, if the message was reordered, it was pushed back by one or two positions in the serial order. In the present case (for the restart setting) 100% of the messages were shuffled by one or two positions.

The delegate server collects responses from all servers, so the observed response time is the shortest response time of all servers. This means that if a restart occurs, it will probably have no effect on the response time, as the response will come from a server were no restart did occur. So the influence of a restart can only be measured on the overall response time if the restart occurs on all servers. If we consider that out of order messages (and therefore transaction restarts) are independent random events for each server, the chances of a restart occurring on all servers is quite low. Therefore, in order to measure of the response time of restarted transaction, we have set the simulator to deliver messages out of order on all servers at the same time.

We see that when the conditions are good (no restarts) the optimistic algorithm outperforms the classical algorithm by around 7 *ms*. Using the optimistic algorithm, transaction processing starts after the initial broadcast, while in normal active replication, processing starts after the end of the total order broadcast protocol. The difference should therefore amount for two broadcasts, and one round of

point-to-point messages. Given the parameters of the simulator, and considering that there is little network contention, both a point to point message and a broadcast take 1.5 *ms* (0.5 *ms* at the sender, 0.5 *ms* on the wire and 0.5 *ms* on the receiver). The difference between optimistic and active replication should therefore amount to $2 \times 1.5 + 1 \times 1.5 = 4.5$ *ms*. The observed difference is quite close to that.

We also see that the performance of the optimistic algorithm is not good when all messages are out of order. We will show in the next paragraph that is not due to restarted transaction, but to the increased lock contention and degraded synchronisation. It is interesting to note that the difference in response time between the two settings of the optimistic technique (with restarts and without) is much more important when the load is low. This is probably due to the fact that as transactions are delivered at a lower rate, conflicts are detected later and so the "cost" of a restart, in terms of processing, is higher. In fact, the response time of restarted transactions was more or less stable at around 180 *ms*, regardless of the load.

Restart Rates, Restart Times. We see that the optimistic technique is good when the spontaneous ordering property holds, but not so good when it does not. To understand the influence of out of order messages, we fixed the load at 40 transactions per second and changed the proportion of out of order messages. During this experiment, we measured the following data:

Redo-Rate. The proportion of transactions that needed to be restarted.

Response Time. The mean response time of the overall system.

Redo Response Time. The mean response time of transactions that were restarted because of out of order messages.

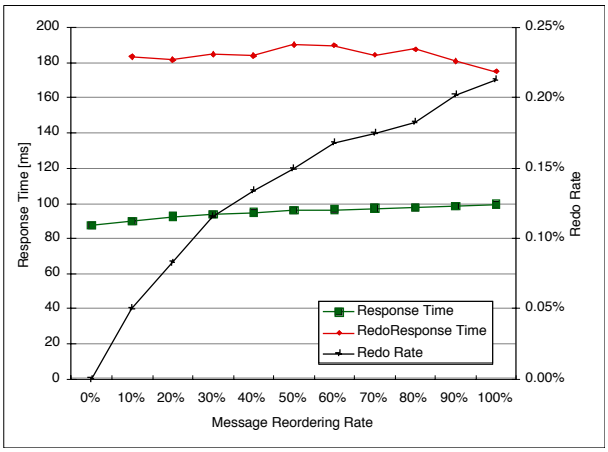

Figure 5.23: Response time of restarted transactions (optimistic active replication)

Figure 5.23 shows the result of this experiment with three curves: one for the overall response time, one for the response time of restarted transactions and one for the transaction restart rate. The left Y axis represents the response time, expressed in milliseconds, and is used for response time curves. The right Y axis represents the restart rate of transactions, i.e. the proportion of transactions that needed to be restarted. The redo response time has no point when $X = 0$, as there are no restarts.

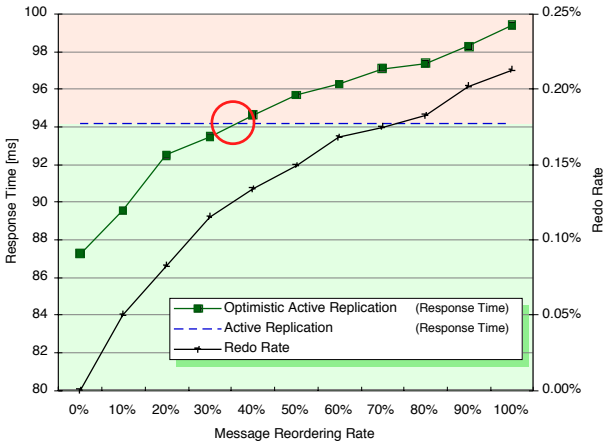

Figure 5.24: Out of order message influence on overall performance

Figure 5.24 shows a close up of the response time of the optimistic technique compared to response time of the active replication technique. The restart proportion is also plotted. The X axis represents the proportion of out of order messages. The left Y axis represents again the response time and was zoomed to the range between 80 and 100 *ms*. The right Y axis represents the proportions of restarted transactions. When $X = 0$ the response time corresponds to the response time of "no restarts" curve in Figure 5.22, when $X = 1.0$, the response time corresponds to the response time of the "restarts" curve in the same figure.

As expected, we see that the number of restarted transactions increases with the number of out of order messages, still this rate stays low (less than 0.25%) even when all messages are out of order. We also see that the overall response time increases with the number of out of order messages. The interesting thing is that the increase of the response time cannot be explained simply by the increased number of restarted transactions: overall this contribution would be negligible (0.25% of the transactions with doubled response time would increase the mean response time by less than 1%). The increase of the mean response time is mostly due to higher lock contention and synchronisation costs.[5] At some point, the optimistic technique is out performed by the classical active replication technique. This point

[5]When a transaction is delivered out of order and restarted, all the locks it held before being aborted delayed other transactions.

is reached when more than 30% of the messages are out of order (circle in Figure 5.24).

Restarted transactions have roughly a response time that is twice as large as the response time of non-restarted transactions. Traditionally, it is assumed that restarted transactions would have a response time that is 50% higher than non-restarted transactions, as a transaction is restarted in average in the middle of its execution. This is not the case here. The response time of a restarted transaction is 100% because transactions are not restarted immediately when they are aborted, but when the definitive total order is known. This avoids situations were a transaction might restart multiple times[6] and also ensures that the response time is bounded. Contention on the latch used for locking all items in a transaction in one atomic step is also lowered.

5.3.6.C Discussion

Overall, the optimistic active replication technique makes sense if the spontaneous ordering property holds most of the time (70%). If out of messages are very rare (as is the case most of the time in a LAN), the optimisation improves the response time by around 6%. Depending on the condition and the respective performance of transaction processing and communication protocol, this gain can change quite a lot.

Restarts are rare because they only occur when two messages m_1 and m_2 that contain conflicting transactions t_1 and t_2 are delivered in the wrong relative order. If the conflict rate is reasonable, this event is rare. In our setting, the restart rates stayed below 0.25%. If out of order messages occur independently on all servers, then the probability that a client observes the response time of restarted transaction is p^n where p is the probability of a restart on one server, and n the number of servers. When p is low and n is large, this number is very small. In the event that a message is delivered out of order on all servers, the number of restarts is bounded. This is interesting because this means that this replication technique has a stable response time most of the time, and could therefore be used for soft-real time applications.

The added delay of restarted transactions is hidden because we collect the response from all servers. While this increases the network usage, the response time is lowered. Response collection makes sense when the time to send a message between servers is below the standard deviation of the response time.

While the spontaneous ordering property only holds in a LAN setting, optimistic active replication could also be used in a WAN setting, by using some heuristics to build a reasonable tentative order – as long as this tentative order is generally the same as the final delivery order, the impact will not be significant in terms of restarted transactions. Using this technique in a WAN setting makes sense,

[6] If this were not the case, a transaction could potentially restart for each level of out of order delivery. So if a message was 10 positions out of order, then a transaction could restart 10 times.

because optimistic active replication tries to optimise out the time needed to process the total order broadcast protocol. In a LAN setting, the response time of total order broadcast is very low, so optimising it out is probably not worth the trouble. In a WAN the response time of the total order broadcast is large, so optimising it out makes sense. Response collection would not make less sense in a WAN setting: the increased latency means that response from remote servers will always arrive on the delegate after the local response, even if the event of a transaction restart. The impact on bandwidth would also be important.

To summarise, optimistic active replication offers increased performance and is an interesting option when the processing time is roughly the same as the time needed to execute the total order broadcast protocol and the spontaneous ordering property holds. The cost of restarts can be hidden by collecting the response from all servers.

5.4 Conclusion

In this chapter, we measured the performance of different replication techniques. We focused on different group communication-based techniques and compared their performance with classical replication schemes like primary-copy, distributed-locking, and lazy update-everywhere replication. We also measured the performance gain of two optimisations: group-safe certification replication and optimistic active replication.

Ser-D was evaluated in [Kem00] and compared to other techniques, like distributed-locking, but with a more simplified network model. The performance of certification-based replication was evaluated in [Ped99], but not compared to other techniques. This evaluation also relied on a simplified network model and used a database with versioning. Another evaluation of the performance of certification replication using a prototype and a CORBA middleware is presented in [Van00].

Performance evaluations show that group communication-based replication significantly outperforms traditional database replication protocols like distributed-locking. The performance difference is large if the network is slow and subject to contention. This is due to the fact that distributed-locking uses a lot of messaging. Group communication-based replication uses less networking resources by relying on abstractions like total order broadcast. This kind of replication is therefore very efficient with a slow network. When the network is fast, and contention rare, group communication-based replication still outperforms distributed-locking, albeit by a smaller amount – a faster network reduces the cost of such techniques, but does not negate the cost of synchronising all replicas.

The results presented in this chapter confirm the overall statements of previous simulation work [Ped99, HAA99b, Kem00]: group communication-based database replication offers good performance. The results presented here are consistent with those described in [Kem00] that shows the performance of different variants of Ser-D, all outperforming distributed-locking. It is interesting to note that most

variants proposed in [Kem00] offer increased performance by relaxing serialisability and relying instead on cursor stability. In this context, it is worth noting that certification replication outperforms Ser-D replication in most cases, while still enforcing serialisability. The results are also consistent with the observations about the performance of certification based replication in [Ped99].

The lack of difference between the experiment with the slow and the fast network and the resource usage of the simulator confirms a general observation that in a LAN situation, network contention is not a real problem. Resources like CPU and disks are much more likely candidates for being bottlenecks in a LAN. This does not mean that communication is not an issue in database replication: multiple copies still induce several problems like synchronisations costs and high abort rates. However those issues are not related to the performance of the network, but to the design of the replication scheme. The performance difference between certification-based replication and Ser-D replication is not due to the cost of executing a total order broadcast, but the cost of adding a synchronisation barrier between the replicas.

Chapter 6

Conclusion

That is not dead which can eternal lie,
And with strange æons even death may die.

Abdul Alhazred

6.1 Research Assessment

This research has led to three contributions to the understanding of database replication based on group communication. One contribution is structural, one is qualitative, and one quantitative. The structural contribution is a classification of replication techniques. The qualitative contribution is an exploration of fault-tolerance criterion and their relationship with database replication based on group communication. The quantitative contribution is a performance evaluation.

6.1.1 Classification

A systematic classification of database replication techniques has been proposed. This classification shows the different way databases can be replicated and analyses the requirements for each category of techniques. The requirements are expressed for both the database sub-system (amount of determinism required) and the group communication sub-system (properties of the communication primitives).

The classification highlights the logical structure of replication techniques and shows the relationship between techniques described in the literature. It also shows categories of techniques that have not be explored thoroughly.

6.1.2 Fault-Tolerance Criterion

The relationship between the model of the group communication and the resulting fault-tolerance criterion has been explored. We show that 2-safe replication based on group communication is not possible with current group communication implementations. This shows the limitation in the way group communication systems are designed and how their interfaces are specified when considering applications that, like database systems, can recover their state.

Group-safety. An alternative safety criterion, *group-safety*, is proposed. This criterion is more suited for group communication-based replication techniques than the classical 1-safety or 2-safety criterion. Group-safe database replication techniques represents a middle-ground between 1-safe and 2-safe database replication. A 1-safe replication technique can lose transactions when one replica crashes; in a 2-safe technique, all n replicas may crash, and no transaction will be lost. Group-safe replication can withstand f ($0 < f < n$) failures without losing a transaction.

Group-safe replication. Group-safety implies a shift in the way the durability aspect of the ACID properties is considered. In a group-safe technique, durability is not ensured with stable storage, like in traditional databases. Instead the *group* is responsible of ensuring that a transaction's effect is durable. This shift is possible because a replicated database has basically two fault-tolerance mechanisms: stable-storage and replication. Transferring the responsibility from one component (the disks) to another (the group of servers) makes sense for performance reasons: in a LAN network access is much faster than disk access.

6.1.3 Performance Evaluation

Performance evaluation offers an insight of the performance and scalability of group communication-based replication techniques. The results confirm that group communication-based replication can offer good performance and very good scalability.

Group communication-based Database Replication. Experiment show that group communication-based replication offers a good alternative to distributed-locking database replication. The different is very important in presence of a slow network, but remains apparent even if the network is fast.

Active Replication. The experiment with the active replication technique showed that in low-load setting, the group of servers could offer improved performance, giving the client the impression of having always the best latency. Active replication can also be improved using optimistic delivery of the total order broadcast primitive.

Certification and Ser-D techniques The evaluation of the two group communication-based techniques that were designed for database replication (Ser-D and certification) show that those technique are very interesting from the performance point of view. The comparison of the Ser-D and the certification technique show that while communication costs are very low in a LAN, synchronisation always has an impact on performance. This comparison shows that the performance of the Ser-D technique can be improved even without relaxing serialisability. Experiments also showed that the kind of certification algorithm has a large impact on the scalability of the certification technique.

Group Safe Replication. The group-safe version of the certification technique showed very impressive results, outperforming even lazy replication. This confirms that group-safe replication is an interesting alternative to lazy replication. Group-safe replication offers good performance and strong fault-tolerance.

6.2 Open Questions and Future Research Direction

Besides the contributions presented in the previous section, this research has raised many interesting questions and issues, that deserve further research.

6.2.1 Intra-Layer Communications

The problem of 1 and 2-safety has led us to analyse the communications between the application and the group communication layer in terms of message exchange. This approach could be used to specify in a clear way group communication primitives. This would help unify all the variants of group communication primitives that use different flow controls and calling semantics [DSS98, Boi01]. Such a tool would also help design new variants of group communication primitives. A clear specification of group communication primitives would also give a clearer view of the interface between application and group communication layer, and permit a better design of both components.

The *ack* messages also highlight how, when both the group communication and application layer have stable storage, data and the responsibility (for fault-tolerance) gets transferred from one component to the other. This notion could be extended to a context where the transport layer also has access to stable storage, for instance if message queues are used for sending messages [MS97]. This raises the question of end-to-end durability: how to ensure durability in a n-tier architecture when certain elements have access to stable storage, and others do not?

6.2.2 Hybrid Replication techniques

Some of the possible replication techniques described in the classification have not been implemented – most of them use the total order broadcast primitive to

avoid deadlocks but still rely on a voting phase to ensure atomicity. Those "hybrid" techniques are interesting because they mix the mechanisms used in voting and non-voting replication. They rely on total order broadcast to minimise deadlocks and network communication, but do not require the strict determinism, like other group communication-based replication techniques. Replicas can act in a non-deterministic way and unilaterally abort transactions, the atomic commitment ensures that all replicas end up with the same state.

6.2.3 Best-Effort Total Order Broadcast

Hybrid techniques, but also optimistic active replication, benefit from a very fast, best-effort total order broadcast implementation. The spontaneous ordering property can be used to build one such implementation, but many others are certainly possible – one area of particular interest would be best-effort total order broadcast in an MAN settings, where group communication-based replication would make a lot of sense and spontaneous ordering property would not hold.

6.2.4 Group-Safe Replication

The group-safe version of the certification technique showed very good performance, but this performance could certainly be improved with other optimisations. One possible approach would be to use a middle-tier to route certain transactions to certain replicas, thus optimising the caching behaviour. In group-safe replication, writes are executed outside of the scope of the transaction and therefore, have no direct impact on the response time. This means that read operations are responsible for a large part of the response time, so optimising the performance of read operations by using improved caching makes a lot of sense.

6.2.5 Optimistic Mechanisms

The optimistic active replication scheme presented in Chapter 5 represents only one possible use early delivery of messages. One drawback of this technique is its complexity: the whole transaction handling policy needs to be changed to take advantage of early delivery. One interesting use of early delivered messages would be to control caching: when a message is delivered in tentative order, the transaction it contains is parsed and the relevant items loaded in the cache. Such a design would be interesting because it couples two optimistic designs: early delivery and caching. A wrong spontaneous ordering requires no special care – if messages are tentatively delivered in the wrong order, some items might not be preloaded in the cache.

Another opportunity for optimism would be in techniques that replicate high-level transactions [PF00]. In those techniques the high-level transactions are transformed in low-level transactions on each replica. This transformation could be started upon optimistic delivery and the result cached. Upon total order delivery,

the cached version is used. In case of out of order early delivery, the content of the cache is discarded. This scheme would be very interesting for WAN replication schemes [FP01], where the time difference between optimistic delivery and total-order delivery could be quite important.

Bibliography

[AAES97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), 1997.

[ACL87] R. Agrawal, M. J. Carey, and M. Livny. Concurency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[ACT97] M. K. Aguilera, W. Chen, and S. Toueg. Quiescent reliable communication and quiescent consensus in partitionable networks. Technical Report TR97-1632, Cornell University, Computer Science Department, June 1997.

[ACT00] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash recovery model. *Distributed Computing*, 13(2):99–125, April 2000.

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash recovery model. In *Proceedings of the* 12^{th} *International Symposium on Distributed Computing (DISC'1998 fomerly WDAG)*, volume 1499 of *Lecture Notes in Computer Science*, pages 231–245, Andros, Greece, September 98. Springer Verlag.

[Alo97] G. Alonso. Partial database replication and group communication primitives (extended abstract). In *Proceedings of the* 2^{nd} *European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 171–176, Zinal (Valais, Switzerland), January 1997.

[ANS92] American National Standart for Information Systems, 1819 L Street, NW, Washington, DC 20036, USA. *ANSI X3.135-1992 – Database Language SQL*, November 1992.

[BAD+84] K P. Birman, A. El Abbadi, W. Dietrich, T. A. Joseph, and T. Raeuchle. An overview of the Isis project. Technical Report

TR84-642, Cornell University, Computer Science Department, October 1984.

[BBC+98] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman. The Asilomar report on database research. Technical Report MSR-TR-98-57, Microsoft Research, One Microsoft Way, Redmond, WA 98052, September 1998.

[BCBT96] A. Basu, B. Charron-Bost, and S. Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR06-1609, Cornell University, Computer Science Departement, Ithaca NY 14853 USA, September 1996.

[BCH+98] A. Baratloo, P. E. Chung, Y. H. Huang, S. Rangarajan, and S. Yajnik. Filterfresh: Hot replication of java rmi server objects. In *Proceedings of the* 4^{th} *Conference on Object Oriented Technologies and Systems (COOTS)*, pages 59–63, Santa Fe, New Mexico, USA, 1998. USENIX.

[BCH+00] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00)*, Hilton Head, South Carolina USA, January 2000.

[BFG01] R. Boichat, S. Frølund, and R. Guerraoui. Open consensus. *Concurency and Computation: Practice and Experience*, 2001.

[BGHJ92] A. Bhide, A. Goyal, H. Hsiao, and A. Jhingran. An efficient scheme for providing high availability. In *Proceedings of 1992 SIGMOD International Conference on Management of Data*, pages 236–245, May 1992.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–239, June 1992.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJ87] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the* 11^{th} *ACM Symposium on OS Principles*, pages 123–138, Austin, TX, USA, November 1987. ACM SIGOPS, ACM.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *Distributed Systems*, chapter 8 – The primary-backup approach, pages 199–216. ACM Press. Addison-Wesley, second edition, July 1993.

[BO83] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 27–30. ACM, 1983.

[Boi01] R. Boichat. *Reliable and Total Order Broadcast in the Crash Recovery Model*. PhD thesis 2472, École Polytechnique Fédérale Lausanne, Switzerland, November 2001.

[BSR80] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie. Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 5(1):18–51, March 1980.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[BT93] Ö. Babaoğlu and S. Toueg. Understanding non-blocking atomic commitement. Technical Report UBLCS-93-2, Laboratory for Computer Science, University of Bologna, 5 Piazza di Porta S. Donato, 40127 Bologna (Italy), January 1993.

[BT00] R. Bergman and C. Tsounis. DB2 univeral data base version 7 features and facilities. White paper, IBM Corporation, April 2000.

[Cap90] C. H. Cap. Distributed systems with data replication: A non-technical survey. Technical Report ifi-90.11, Department of Computer Science, University of Zürich, 190, Winterthurstraße CH-8057 Zürich, Switzerland, November 1990.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Doley. Atomic broadcast: From simple message diffusion to byzantine agreement. *Proceedings of the* 15^{th} *Internationnal Conference on Fault-Tolerant Computing*, pages 1–12, June 1985.

[CBDS01] Bernadette Charron-Bost, Xavier Défago, and André Schiper. Time vs. space in fault-tolerant distributed systems. In *Proceedings of the* 6^{th} *International Workshop on Object-oriented Real-time Dependable Systems (WORDS'01)*, Rome, Italy, January 2001. IEEE, IEEE Computer Society.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[Cha84] J. M. Chang. Simplifying distributed database systems design by using a broadcast network. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting*, volume 14, pages 223–233, Boston, Massachusetts USA, 18–21 June 1984. ACM, Special Interest Group on Management of Data, New York, NY, USA.

[CHKS94] S. Ceri, M. Houtsma, A. Keller, and P. Samarati. A classification of update methods for replicated databases. Technical Report CS-TR-91-1392, Stanford University, Computer Science Departement, May 1994.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[CP92] S. W. Chen and C. Pu. A structural classification of integrated replica control mechanisms. Technical Report CUCS-006-92, Columbia University, Departement of Computer Science, New York, NY 10027, 1992.

[CS93] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In Barbara Liskov, editor, *Proceedings of the* 14^{th} *Symposium on Operating Systems Principles*, volume 27, pages 44–57, Asheville, North Carolina, December 1993. ACM Press, New York, NY, USA.

[CSAH98] P. K. Chysanthis, G. Samaras, and Y. J. Al-Houmaily. *Recovery Mechanisms in Database Systems*, chapter Recovery and Performance of Atomic Commit Processing in Distributed Database Systems. Prentice-Hall, first edition, 1998.

[CT92] S. Chen and D Towsley. A performance evaluation of RAID architectures. Technical Report UM-CS-1992-067, Departement of Computer Science, University of Masschusetts, Amherst, MA 01003 USA, September 1992.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CZ85] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[Déf00] X. Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis 2229, École Polytechnique Fédérale de Lausanne, Switzerland, August 2000.

[DGMS85] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[DM96] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the* 17^{th} *Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998. IEEE.

[DWAP94] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, Monterey, California, November 1994. USENIX Association. Also appeared as University of California Technical Report CSD-94-844.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976. Also published in/as: IBM Research Report RJ1487, San Jose, CA, December, 1974.

[FC94] A. W. Fu and D. W. Cheung. A transaction replication scheme for a replicated database with node autonomy. In *Proceedings of the International Conference on Very Large Databases*, Santiago, Chile, 1994.

[Fel98] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis 1867, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

[FLP85] M. H. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FP01] S. Frølund and F. Pedone. Continental pronto. In *Proceeding of the* 20^{th} *Symposium on Reliable Distributed Systems*, pages 46–55, New Orleans, LA, USA, October 2001. IEEE Computer Society, Los Alamitos, California, USA.

[Gär01] F. C. Gärtner. A gentle introduction to failure detectors and related problems. Technical Report TUD-BS-2001-01, Darmstadt University of Technology, Department of Computer Science, April 2001.

[GHOS96] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996. ACM-SIGMOD.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles SOSP 7*, pages 150–162, Asilomar Conference Grounds, Pacific Grove CA, December 1979. ACM, New York.

[GMP90] H. Garcia-Molina and C. A. Polyzois. Two epoch algorithms for disaster recovery. In *Proceedings of* 16^{th} *VLDB Conference*, pages 222–230, Brisbane, Australia, 1990.

[GNSY00] A. Gokhale, B. Natarajan, D. C. Schmidt, and S. Yajnik. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the* 2^{nd} *International Symposium on Distributed Objects and Applications (DOA '00)*, Antwerp, Belgium, September 2000. Object Management Group.

[Gol94] R. Goldring. A discussion of database replication technology. *Info DB*, 1(8), May 1994.

[Gol95] R. Goldring. Things every update replication customer should know (abstract). In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 439–440, San Jose, CA USA, May 1995.

[GR93] J. N. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[GS97] R. Guerraoui and A. Schiper. Genuine atomic multicast. In *Proceedings of the* 11^{th} *International Workshop on Distributed Algorithms (WDAG-11)*, Saarbrücken, Germany, September 1997.

[Gue95] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proceedings of the* 9^{th} *International Workshop on Distributed Algorithms (WDAG-9)*, LNCS 972, pages 87–100, Le Mont-St-Michel, France, September 1995. Springer-Verlag.

[HAA99a] J. Holliday, D. Agrawal, and A. El Abbadi. Database replication: If you must be lazy, be consistent. In *Proceedings of* 18^{th} *Symposium on Reliable Distributed Systems SRDS'99*, pages 304–305. IEEE Computer Society Press, October 1999.

[HAA99b] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group multicast. In *Proceedings of International Symposium on Fault Tolerant Computing (FTCS29)*, pages 158–165. IEEE Computer Society, 1999.

[HAA00] J. Holliday, D. Agrawal, and A. El Abbadi. Using multicast communication to reduce deadlocks in replicated databases. In *Proceedings of the* 19^{th} *Symposium on Reliable Distributed Systems SRDS'2000*, pages 196–205, Nürnberg, Germany, October 2000. IEEE Computer Society, Los Alamitos, California.

[Had88] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, January 1988.

[Hol01] J. Holliday. Replicated database recovery using multicast communications. In *Proceedings of the Symposium on Network Computing and Applications (NCA'01)*, pages 104–107, Cambridge, MA, USA, October 2001. IEEE.

[HS00] M. A. Hiltunen and R. D. Schlichting. The cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nürnberg, Germany, October 2000.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, second edition, 1993.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Inf98] Informix, 4100 Bohannon Drive, Menlo Park, California 94025 USA. *Informix Replication: A High-Performance Solution for Distributing and Sharing Information*, June 1998.

[Jai91] R. Jain. *The art of computer system performance analysis: techniques for experimental design, measurement, simulation and modeling*. John Wiley and Sons, Inc., New York USA, 1991.

[Jaj99] S. Jajodia. Data replication gaining popularity. *IEEE Concurency*, pages 85–86, April 1999. Interview of Yuri Breitbart and Hank Korth.

[JPPMAA01] R. Jiménez-Paris, M. Patiño-Martínez, G. Alonso, and S. Arévalo. A low latency non-blocking commit server. In J. Welch, editor, *Proceeedings of the* 15^{th} *Internationnal Conference on Distributed Computing (DISC 2001)*, volume 2180 of *lecture notes on computer science*, pages 93–107, Lisbon, Portugal, October 2001. Springer Verlag, Berlin Heidelberg.

[KA98] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the* 18^{th} *International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, May 1998.

[KA99] B. Kemme and G. Alonso. Transactions, messages and events: Merging group communication and database system. In 3^{rd} *Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island (Portugal), April 23–28, 1999. BROADCAST Esprit WG 22455.

[KA00a] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the* 26^{th} *International Conference on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[KB94] N. Krishnakumar and A. J. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Transactions on Database Systems*, 19(4):586–625, December 1994.

[KBB01] B. Kemme, A. Bartoli, and Ö. Babaoğlu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the Internationnal Conference on Dependable Systems and Networks (DSN2001)*, Göteborg, Sweden, June 2001.

[Kei94] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, The Hebrew University of Jerusalem, Jerusalem, Israel, April 1994. also technical report CS94.

[Kem00] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis 13864, Swiss Federal Institute of Technology Zürich, Switzerland, August 2000. No. 13864.

[Kim00] K. H. Kim. Issues insufficiently resolved in century 20 in the fault-tolerant distributed computing field. In *Proceedings of the* 19^{th}

Symposium on Reliable Distributed Systems, pages 106–115, Nürnberg Germany, October 2000. IEEE Computer Society. Invited Paper.

[KLW96] E. Kindler, A. Listl, and R. Walter. A specification method for transaction models with data replication. Technical Report 56, Humboldt-Universität zu Berlin Germany, March 1996.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the International Conference on Distributed Computing Systems*, Austin, Texas, June 1999.

[KS93] A. Kumar and A. Segev. Cost and availability tradeoffs in replicated concurrency control. *ACM Transactions on Database Systems*, 18(1):102–131, March 1993.

[KT91] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the* 11^{th} *International Conference on Distributed Computing Systems ICDCS*, pages 222–230, Washington, D.C., USA, May 1991. IEEE Computer Society Press.

[KT94] M. F. Kaashoek and A. S. Tanenbaum. Efficient reliable group communication for distributed systems. Technical report, M.I.T., Cambridge, and Vrije Universiteit, Amsterdam, 1994.

[Lam89] L. Lamport. The part-time parliament. Technical report, System Research Center Digital Equipement Corp, Palo Alto, USA, September 1989. A revised version was published in [Lam98].

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[LS98] M. C. Little and S. K. Shrivastava. Understanding the role of atomic transactions and group communications in implementing persistent objects. In *Eighth International Workshop on Persistent Object Systems: Design Implementation and Use*, August 1998.

[LSG+79] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Potzulo, and B. W. Wade. Notes on distributed databases. Technical Report RJ2571(33471), IBM, San Jose Research Laboratory, 1979.

[Lyo88] J. Lyon. Design considerations in replicated database systems for disaster protection. In *Proceedings of IEEE Compcon*, 1988.

[Lyo90] J. Lyon. Tandem's remote data facility. In *Proceedings of IEEE Compcon*, 1990.

[Man02] K. Maney. Microsoft shifts its focus to security. *USA Today*, January 2002.

[Mes94] Mesquite Software Inc., 3925 West Braker Lane Austin Texas TX-78759-5321, USA. *CSIM18 simulation engine (C++ version)*, 1994.

[MFSW95] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phœnix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA, October 1995. IEEE. Workshop held during the 7^{th} Symposium on Parallel and Distributed Processing, (SPDP-7).

[MMSA+96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[MPR01] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the* 21^{st} *International Conference on Distributed Computing Systems (ICDCS-01)*, pages 707–710, Phoenix, Arizona, USA, April 2001. IEEE Computer Society.

[MS97] S. Maffeis and D. C. Schmidt. Constructing reliable distributed communication systems with CORBA. *IEEE Communications Magazine*, 14(2):56–61, February 1997.

[Mun98] T. Munk. *DB2 Familly: Client/Server Performance Measurement Series*. IBM, IBM Santa Teresa Laboratory, San Jose, California USA, August 1998.

[Neu02] P. G. Neumann. The risk digest: Forum on risks to the public in computers and related systems. *ACM Committee on Computers and Public Policy*, 21(87), January 2002.

[NMMS99] P. Narasimhan, L. Moser, and P. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *Proceedings of the* 18^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS'99)*, pages 263–273, Lausanne, Switzerland, October 1999. IEEE.

[NSB97] K. Nørvåg, O. Sandstå, and K. Bratbergsengen. Concurrency control in distributed object-oriented database systems. In *ADBIS*, pages 9–17, St. Petersburg (Russia), 1997.

[OMG01] *Common Object Request Broker Architecture (CORBA) version 2.5*, chapter Fault-Tolerant CORBA, pages 25–1 – 25–116. Object Management Group, 250 First Avenue, Suite 201 Needham, MA 02494 USA, September 2001.

[Ora98] Oracle Corporation, 500, Oracle Parkway, Redwoord City, CA 94065. *Oracle8i^{TM}Advanced Replication*, November 1998. Oracle Technical White Paper.

[ÖV99] M. Tamer Özsu and P. Valdurez. *Principles of Distributed Database Systems*. Prentice Hall, Upper Saddle River, New Jersey 07458 USA, second edition, 1999.

[PCD91] D. Powell, M. Chéréque, and D. Drackley. Fault-tolerance in Delta-4*. *ACM Operating Systems Review, SIGOPS*, 25(2):122–125, April 1991.

[Ped99] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis 2090, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[Ped01] F. Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, December 2001.

[PF00] F. Pedone and S. Frølund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of* 19^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, Nürnberg, Germany, October 2000. IEEE Computer Society.

[PG97] F. Pedone and R. Guerraoui. On transaction liveness in replicated databases. In *Proceedings of IEEE Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS'97)*, December 1997.

[PGM89] F. Pittelli and H. Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems*, 7(1):25–60, February 1989.

[PGM94] C. A. Polyzois and H. García-Molina. Evaluation of remote backup algorithms for transaction-processing systems. *ACM Transactions on Database Systems*, 19(3):423–449, September 1994.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the* 16^{th} *Symposium on Reliable Distributed Systems (SRDS-16)*, Durham, North Carolina, USA, October 1997.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, September 1998.

[PGS99] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. Technical Report SSC/1999/008, École Polytechnique Fédérale de Lausanne, Switzerland, March 1999.

[PMJPA01] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Group transactions: An integrated approach to transactions and group communication. In *Workshop on Concurrency in Dependable Computing*, Newcastle Upon Tyne, United Kingdom, June 2001.

[Pol95] S. Poledna. *Fault-Tolerant Real-Time Systems: the Problem of Replica Determinism*, volume 345 of *Enginering and Computer Science*. Kluwer Academic Publishers, Boston, November 1995.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, September 1999.

[PSWL94] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of Arjuna. Technical Report TR94-65, ESPRIT Basic Research Project BROADCAST, October 1994.

[RFV96] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proceedings of the* 16^{th} *International Conference on Distributed Computing Systems (ICDCS'96)*, pages 503–510, Hong Kong, May 1996. IEEE.

[RJP01] G. Alonso R. Jiménez-Peris, M. Patiño-Martínez. Is reliable multicast too expensive? let's be optimistic. In *Proceedings of the* 4^{th} *CaberNet Workshop*, Pisa, Italy, October 2001.

[RR00] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proceedings of the* 20^{th} *International Conference on Distributed Systems (ICDCS'2000)*, pages 288–295, Taipei, Taiwan (ROC), April 2000. IEEE Computer Society, Los Alamitos USA.

[RST95] S. Rangaranjan, S. Setia, and S. K. Tripathi. A fault-tolerant algorithm for replicated data management. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1271–1282, December 1995.

[SAA98] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* 18^{th} *IEEE International Conference on Distributed Computing Systems ICDCS'98*, pages 148–155, Amsterdam, The Netherlands, May 1998. IEEE.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Ske81] D. Skeen. Nonblocking commit protocols. In Y. Edmund Lien, editor, *Proceedings of the 1981 International Conference on Management of Data*, pages 133–142, Ann Arbor, Michigan USA, April 1981. ACM SIGMOD, New York.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[Sta94] D. Stacey. Replication: DB2, Oracle, or Sybase. *Database Programming & Design*, 7(12), 1994.

[Sta95] D. Stacey. Replication: DB2, Oracle, or Sybase. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(4):95–101, December 1995.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5:188–194, May 1979.

[SW99] R. Schenkel and G. Weikum. Experiences with building a federated transaction manager based on CORBA OTS. In *Proceedings of the 2nd Workshop EFIS'99*, pages 79–94, Kühlungsborn, Germany, May 1999. Infix, Sankt Augustin.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[TP98] O. Theel and H. Pagnia. Optimal replica control protocols exhibit symmetric operation availabilities. In *Proc. of the Int. Symp. on Fault-Tolerant Computing FTCS*, 1998.

[UDS00] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proceedings of the* 9^{th} *IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, October 2000.

[UDS01] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15)*, Beppu City, Japan, February 2001.

[Van00] R. Vandewall. Database replication prototype. Master's thesis, Rijksuniversiteit Groningen and École Polytechnique Fédérale de Lausanne, Netherlands and Switzerland, August 2000.

[vBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[VCKD99] R. Vitenberg, G. V. Chockler, I. Keidar, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report MIT-LCS-TR-790, Massachusetts Institute of Technology, 77 Massachusetts Avenue Cambridge, MA 02139-4307 USA, September 1999.

[WAL97] Y. M. Wang, O. P. Amani, and W. J. Lee. Reliability issues in distributed component object model (DCOM). In *Proceedings of* 4^{th} *International Workshop on Community Networking (CN4)*, pages 59–63, Atlanta, Georgia, USA, September 1997. IEEE, IEEE. position paper.

[WPS99] M. Wiesmann, F. Pedone, and A. Schiper. A systematic classification of replicated database protocols based on atomic broadcast. In *Proceedings of the* 3^{rd} *Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99)*, pages 264–274, Madeira Island (Portugal), April 23–28, 1999. BROADCAST Esprit WG 22455.

[WPS+00a] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of* 19^{th} *Symposium on Reliable Distributed Systems (SRDS'2000)*, pages 206–215, Nürnberg, Germany, October 2000. IEEE Computer Society.

[WPS+00b] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of* 20^{th} *International Conference on Distributed Computing Systems (ICDCS'2000)*, Taipei, Taiwan, R.O.C., April 2000. IEEE Computer Society Los Alamitos California.

[WS95] U. G. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of the* 14^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS-14)*, Bad Neuenahr, Germany, September 1995.

Curriculum Vitæ

I was born in Winterthur (Switzerland) in 1972. I attended primary and secondary school in Geneva. At this time, I started to fiddle with a Commodore 64 and got interested in computing. In 1991, I obtained a *Maturité Fédérale Latine* in Collège Calvin, in Geneva.

I started to study computer science at the *Centre Universitaire Informatique* in the University of Geneva. From 1995 onwards, I served as a technical assistant and system administrator in different sections of University. My master thesis was about fault-tolerant designs, I obtained my Diploma in 1997.

Since 1998, I have been working in the Distributed System Laboratory (LSR, formerly LSE) at the Swiss Federal Institute of Technology as a research and teaching assistant and a PhD student under the guidance of Professor André Schiper. In 2000 and 2001 I also worked as a teacher in the University of Applied Science in Fribourg.

Index