THE DATABASE STATE MACHINE AND GROUP COMMUNICATION ISSUES

THÈSE N° 2090 (1999)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Fernando PEDONE

Mestre em Ciências da Computação

originaire du Brésil

jury:

Prof. André Schiper, rapporteur
Prof. Gustavo Alonso, corapporteur
Dr Bernadette Charron-Bost, corapporteur
Prof. Rachid Guerraoui, corapporteur
Prof. Jean-Yves Le Boudec, corapporteur

Lausanne, EPFL
1999

Abstract

Distributed computing is reshaping the way people think about and do daily life activities. On-line ticket reservation, electronic commerce, and telebanking are examples of services that would be hardly imaginable without distributed computing. Nevertheless, widespread use of computers has some implications. As we become more depend on computers, computer malfunction increases in importance. Until recently, discussions about fault tolerant computer systems were restricted to very specific contexts, but this scenario starts to change, though.

This thesis is about the design of fault tolerant computer systems. More specifically, this thesis focuses on how to develop database systems that behave correctly even in the event of failures. In order to achieve this objective, this work exploits the notions of data replication and group communication. Data replication is an intuitive way of dealing with failures: if one copy of the data is not available, access another one. However, guaranteeing the consistency of replicated data is not an easy task. Group communication is a high level abstraction that defines patterns on the communication of computer sites. The present work advocates the use of group communication in order to enforce data consistency.

This thesis makes four major contributions. In the database domain, it introduces the Database State Machine and the Reordering technique. The Database State Machine is an approach to executing transactions in a cluster of database servers that communicate by message passing, and do not have access to shared memory nor to a common clock. In the Database State Machine, read-only transactions are processed locally on a database site, and update transactions are first executed locally on a database site, and them broadcast to the other database sites for certification and possibly commit. The certification test, necessary to commit update transactions, may result in aborts. In order to increase the number of transactions that successfully pass the certification test, we introduce the Reordering technique, which reorders transactions before they are committed.

In the distributed system domain, the Generic Broadcast problem and the Optimistic Atomic Broadcast algorithm are proposed. Generic Broadcast is a group communication primitive that allows applications to define any order requirement they need. Reliable Broadcast, which does not guarantee any order on the delivery of messages, and Atomic Broadcast, which guarantees total order on the delivery of all messages, are special cases of Generic Broadcast. Using Generic Broadcast, we define a group communication primitive that guarantees the exact order needs of the Database State

Machine. We also present an algorithm that solves Generic Broadcast. Optimistic Atomic Broadcast algorithms exploit system properties in order to implement total order delivery fast. These algorithms are based on system properties that do not always hold. However, it they hold for a certain period, ensuring total order delivery of messages is done faster than with traditional Atomic Broadcast algorithms. This thesis discusses optimism in the implementation of Atomic Broadcast primitives, and presents in detail the Optimistic Atomic Broadcast algorithm. The optimistic broadcast approach presented in this thesis is based on the spontaneous total order message reception property, which holds with high probability in local area networks under normal execution conditions (e.g., moderate load).

Résumé

Les systèmes répartis sont en train de modifier profondément nos activités quotidiennes: réservation de billets en-ligne, commerce électronique, telebanking, sont des exemples de services qui n'étaient pas imaginables avant l'arrivée des systèmes répartis. Néanmoins, l'utilisation à grande échelle de systèmes informatiques n'est pas sans conséquence. Plus l'on devient dépendent des ordinateurs, plus leur défaillance pose des problèmes. Jusqu'à récemment, les discussions sur la défaillance des systèmes informatiques ne concernaient que des cercles restreints. La situation est en train d'évoluer.

Cette thèse aborde le problème de la conception de systèmes tolérants aux pannes. Plus spécifiquement, ce travail se concentre sur le développement de bases des données qui se comportent correctement même en cas de défaillances. Pour atteindre ce but, cette thèse se base sur les notions de réplication de données et sur les communications de groupes. La réplication de données est une idée naturelle pour tolérer les pannes: si une copie d'une donnée n'est pas disponible, il suffit d'accéder à une autre copie. Par contre, garantir la cohérence des données répliquées n'est pas une tâche simple. La thèse propose l'utilisation des mécanismes de communication de groupes pour garantir la cohérence des données.

La thèse comporte quatre contributions majeures. Dans le domaine des bases de données, elle introduit la "Database State Machine" et la technique de réordonnancement. La Database State Machine est une manière de gérer des transactions s'exécutant sur un cluster de serveurs de bases de données communiquant par échange de messages, et n'ayant accès ni à une mémoire partagée ni à une horloge commune. Dans ce contexte, les transactions de lecture sont exécutées localement sur un serveur, et les transactions de mise à jour sont d'abord exécutées localement sur un serveur avant d'être diffusées aux autres serveurs pour le test de certification et la validation (commit) éventuelle. Le test de certification, nécessaire à la validation, peut conduire à avorter une transaction. Dans le but d'augmenter le taux de transactions que passent le test de certification, la thèse introduit la technique de réordonnancement, qui réordonne les transactions avant de les certifier.

Dans le domaine de systèmes répartis, le problème de la Diffusion Générique (Generic Broadcast) et l'algorithme de Diffusion Atomique Optimisite (Optimistic Atomic Broadcast) sont introduits. La Diffusion Générique est une primitive de communication de groupes qui permet aux applications de définir l'ordre dont elles ont besoin. La Diffusion Fiable (Reliable Broadcast) qui ne garantit aucun ordre entre les mes-

sages, et la Diffusion Atomique (Atomic Broadcast) qui garantit l'ordre total pour la livraison de messages, sont des cas particuliers de la Diffusion Générique. La Diffusion Générique est une primitive de communication de groupes qui permet d'offrir l'ordre exact nécessaire pour la Database State Machine. La thèse présente également un algorithme qui résout la Diffusion Générique. Les algorithmes de Diffusion Atomique Optimiste exploitent les propriétés du système pour délivrer efficacement les messages dans un ordre total. Ces algorithmes sont basés sur des propriétés du système qui ne sont pas toujours satisfaites. Néanmoins, si elles sont satisfaites durant une certaine période de temps, l'algorithme assure l'ordre total plus efficacement que les algorithmes de diffusion atomique traditionnels. La thèse discute l'optimisme dans le contexte de la mise en oeuvre de la Diffusion Atomique, et présente en détail un algorithme. L'optimisme exploité par cet algorithme est basé sur la propriété d'ordre spontanée, qui est satisfaite avec une probabilité élevée dans des réseaux à petite échelle dans des conditions d'exécution normale (trafic modéré, par exemple).

To My Parents,
for their love and support

To Pascale,
for being such a great person

Acknowledgements

I am very grateful to several people who one way or the other contributed to this work. Firstly, I wish to thank André Schiper for accepting me in his research group, for supervising my work with great care and enthusiasm, and for expressing his confidence in me.

I wish to express my gratitude to Rachid Guerraoui for the valuable scientific discussions we had, and for motivating and encouraging my work.

I wish to thank all my colleagues at the Operating Systems Laboratory who helped me shape my understanding of distributed systems. Specially, I wish to thank, in alphabetical order, Benoît Garbinato, Xavier Défago, Assia Doudou, and Rui Oliveira for their friendship.

I am grateful to Matthias Wiesmann, Gustavo Alonso, and Bettina Kemme for our fruitful collaboration in the Dragon project, and for helping me understand the problems involved in the design of distributed databases.

I am thankful to Prof. Claude Petitpierre for accepting to preside the exam, and to the members of the jury, Dr Bernadette Charron-Bost, and Prof. Jean-Yves Le Boudec for the time they have spent examining this thesis.

Several other people played important roles in my doctorate. Among them, my warm thanks to Kristine Verhamme for her crucial help whenever it was necessary, and to Nelson Duarte and Cláudio Geyer, without whom I would probably not have been able to start my doctorate.

Last, but definitely not least, I am extremely thankful to my family for their love and for being such a positive influence in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> It all depends on how we look at things,
> and not on how they are themselves.
>
> **Carl Jung**

Distributed computing has become an ubiquitous technology in the world. From global to local area networks, distributed computing seems to be everywhere. Computer specialists point out two reasons for that. Firstly, manufacture improvements and large scale production have reduced the cost and increased the performance of computers. Secondly, advances in communication systems have resulted in cheap and fast data transmission, allowing to connect virtually every two computers in the world.

It is early to precisely assess how the computer revolution will impact our society, but some of its effects can already be noticed. On-line ticket reservation, electronic commerce, and telebanking are examples of services that would be hardly imaginable without computers. Nevertheless, widespread use of computers has some implications. As more applications, and people, depend on computers, computer malfunction becomes critical. Until recently, discussions about fault tolerant computer systems were restricted to very specific contexts. This picture starts to change, though.[1]

This thesis is about the design of fault tolerant computer systems. More specifically, this thesis focuses on how to develop database systems that behave correctly even in the event of failures. In order to achieve this objective, this work exploits the notions of data replication and group communication. Data replication is an intuitive way of dealing with failures: if one copy of the data is not available, access another one. However, guaranteeing the consistency of replicated data is not an easy task. Group communication is a high level abstraction that defines patterns on the communication of computer sites. The present work advocates the use of group communication in

[1] Apart from being a historical landmark, the change of the millennium has contributed to enlarge the discussions about the effects of computer failures on human lives.

order to enforce data consistency.

1.1 Replicated Databases

Despite the fact that database replication has been an active area of research since the late 70's [Gif79, Sto79, Tho79], the problem of designing database replication protocols that provide good performance and strong data consistency is still far from having a definitive answer. One reason for this fact is that methods to handle replication designed in the 80's have been shown to perform poorly as the number of replicated database sites increases [GHOS96]. Protocols developed with centralised settings in mind (e.g., two-phase locking), when implemented in a distributed system, have been shown to present excessive synchronisation costs, and rapid growth of distributed deadlocks with the number of database sites.

However, requirements of current applications have increased the demand for high-performance and high-availability databases [Jaj99], resulting in the emergence of new mechanisms to support database replication. Commercial database companies, for example, have focused on solutions that provide weak consistency guarantees [Sta95, Jac95]. Nevertheless, weak guarantees are not intuitive and difficult to use. Furthermore, in many cases, user intervention is necessary to bring the database back to a consistent state. Weak consistency guarantees may become attractive in the future, but so far, they lack the theoretical background that allows for strict protocol specifications and rigorous correctness proofs [BHG87, Jaj99, Pap79].

Transactions are the unit of work of databases [GR93]. Ensuring data consistency in replicated databases comes down to guaranteeing that transaction properties are ensured, independently of the number of database replicas and the way data is distributed among them. Transactions are characterised by the ACID properties: Atomicity, Consistency, Isolation, and Durability [GR93]. The *Atomicity* property states that either all transaction operations are executed or none is. *Consistency* establishes that a transaction is a correct transformation of the state. *Isolation* affirms that even though transactions may execute concurrently, it appears to each transaction that it executes alone. *Durability* states that once a transaction completes successfully, its changes to the state survive failures.

1.2 Group Communication

In the context of client-server distributed systems, the mid-80's and 90's saw the emergence of replication protocols based on group communication. Roughly speaking, group communication gathers processes in sets and provides communication primitives enabling to address sets as individual entities [HT93]. Group communication has received increasing attention in the past years from both practical and theoretical viewpoints. The best known group communication system is Isis [BSS91], which is considered by many as the first system in which the feasibility of the group communication approach was demonstrated. Furthermore, current trends in middle-

ware systems seem to confirm the important role played by group communication primitives [Gro98]. From the theoretical point of view, a sound theory underlying group communication has been developed, and minimal conditions under which group communication primitives are proved to be fault-tolerant have been formally identified [CT96, CHT96].

Group communication primitives can have various semantics, and in particular, they can guarantee causal, atomic, and total order message delivery [BSS91]. For example, Atomic Broadcast, the group communication primitive exploited in this thesis, enables to send messages to a set of processes, with the guarantee that the destinations agree on the set of messages delivered, a property known as *Agreement*, and on the order according to which the messages are delivered, a property known as *Total Order* [HT93]. Atomic Broadcast has been shown to guarantee correct propagation of requests in some distributed system replication techniques [Sch90].

Replication based on group communication has mostly concentrated on client-server distributed computing [GS97]. More recently, some authors have suggested using group communication to develop database replication protocols (e.g., [SR96]). Indeed, similarities between ACID properties and Atomic Broadcast properties suggest that there might be a relation between these two subjects. For example, the Agreement property of Atomic Broadcast can be associated with the Atomicity property of transactions, and the Isolation property of transactions can be associated with the Total Order property of Atomic Broadcast.

1.3 About this Research

This thesis started with the broad objective of investigating the use of group communication primitives to develop database replication protocols in the context of the DRAGON[2] project, a joint effort between the Swiss Federal Institute of Technology in Lausanne (EPFL) and the Swiss Federal Institute of Technology in Zurich (ETHZ). As this work evolved, it turned out that looking at database replication protocols from the viewpoint of distributed systems, and looking at group communication primitives from the viewpoint of distributed databases was, *per se*, an interesting way of approaching two different domains.

1.3.1 Research Objectives

The primary goal of this work is to investigate how group communication can be used to implement database replication protocols. The scope of this research focused on an architecture based on a cluster of database sites. Database sites do not have access to shared memory or a global clock, and communicate through message passing. Users should have the impression that the database cluster is a high-performance and high-availability centralised database site. Therefore, data consistency is mandatory.

[2] DRAGON stands for Database Replication based on Group Communication. DRAGON is funded by the Swiss Federal Institute of Technology (EPFL and ETHZ).

A secondary goal of this research is to better understand the impact of group communication on databases, and database replication protocols on distributed system mechanisms. For example, group communication evolved essentially to handle process replication [BSS91]. Naturally, one may wonder whether a different context will change the way group communication has been defined, and is usually implemented.

1.3.2 Research Contributions

This thesis provides four major contributions. In the database domain, it presents the Database State Machine and the Reordering technique. In the distributed systems domain, this work introduces the Generic Broadcast problem, and the Optimistic Atomic Broadcast algorithm.

Database State Machine. The *Database State Machine* is a database replication approach that defines the way transactions are executed by database sites, and the way database sites interact with each other to commit transactions. In the Database State Machine, transactions are executed locally on a database site according to the two-phase locking concurrency control mechanism, which enforces local data consistency. In order to guarantee global data consistency, database sites interact by means of an Atomic Broadcast primitive, which is the only communication mechanism used by database sites. The requirements that database sites have to meet in this context are discussed in detail. Experimental results show that the Database State Machine is a promising approach to executing transactions in a cluster on database sites.

Reordering Technique. Global data consistency in the Database State Machine relies on some sort of optimistic concurrency control mechanism, called *certification test*. According to this schema, a transaction that requests a commit operation does not have a guarantee that it will be committed, since it may fail the certification test. The Reordering technique is a way of executing the certification test that increases the chances that transactions are committed. Roughly speaking, the Reordering technique exploits characteristics of serial executions and rearranges transactions before they are committed. The Reordering technique has been positively evaluated using a simulation model.

Generic Broadcast. Generic Broadcast is a group communication primitive that allows applications to define order requirements based on a conflict relation. Reliable Broadcast, which does not guarantee any order on the delivery of messages, and Atomic Broadcast, which guarantees order on the delivery of all messages, are special cases of Generic Broadcast. It turns out that for several applications, like the Database State Machine, Reliable Broadcast offers a semantic that is too weak to guarantee correctness. Conversely, Atomic Broadcast offers a semantic that is too strong. Using Generic Broadcast, we can define a group communication primitive that is stronger than Reliable Broadcast, and weaker than Atomic Broadcast. An algorithm that implements Generic Broadcast is presented. In order to compare the

implementations of various group communication primitives, the delivery latency parameter is introduced.

Optimistic Atomic Broadcast. The Optimistic Atomic Broadcast algorithm exploits system properties in order to deliver messages fast. The algorithm is optimistic in the sense that it assumes properties that do not always hold. However, it they hold for a certain period, guaranteeing total order of messages is done faster than with traditional Atomic Broadcast algorithms. This thesis discusses optimism in the implementation of Atomic Broadcast primitives, and presents in detail the Optimistic Atomic Broadcast algorithm. The system property exploited by Optimistic Atomic Broadcast is the *spontaneous total order* property which states that, in some networks, it is highly probable that messages are received at their destinations in the same total order. The spontaneous total order property holds with high probability in local area networks under normal execution conditions (e.g., moderate load).

1.3.3 Thesis Organisation

The thesis is organised as follows. Chapter 2 discusses system models, defines fault-tolerant broadcast and related problems, and formalises some database notations used throughout this thesis. Chapter 3 introduces the Database State Machine and the Reordering technique. The Database State Machine is first analysed by means of a simple probabilistic model, and then by means of a simulation model. Chapter 4 presents the Generic Broadcast problem, shows how it can be used to define a broadcast primitive weaker than Atomic Broadcast, but that still ensures the order needs of the Database State Machine, and presents an algorithm that solves Generic Broadcast. Chapter 5 discusses how Atomic Broadcast algorithms can take advantage of optimistic system assumptions, and presents in detail the Optimistic Atomic Broadcast algorithm. In Chapter 6, we summarise the major results of this work and outline future research directions.

Chapter 2

System Models and Definitions

> A theory has only the alternative of being right or wrong.
> A model has a third possibility: it may be right, but irrelevant.
>
> **Manfred Eigen**

A system model describes precisely and concisely all the hypothesis and important aspects about the system. A model should be as general as possible, to extend the applicability of the results stated (this is typically the case when the results are in the form of impossibility proofs), and compact, to leave out irrelevant details and simplify the approach to the problem. In this chapter, we recall system models considered in the literature and used in this thesis, define the properties of fault-tolerant problems of interest for this work, and present some important database definitions.

2.1 Model Definitions

Distributed system models usually centre their definitions around two basic abstractions: processes and communication channels. In the following, we present some common ways of modelling these abstractions.

2.1.1 Process Model

We characterise processes according to four criteria: synchronisation aspects, mode of failure, information storage, and process membership.

Synchronisation aspects. According to synchronisation aspects, processes can be *synchronous* or *asynchronous*. If processes are asynchronous, then there is no bound on the time necessary to execute a step. By contrast, if processes are synchronous, there exists a known bound on their relative speed, that is, for some known bounded number of steps taken by any process, every other process takes at least one step.

Throughout this work, we consider that processes are asynchronous, and to simplify the presentation, we assume the existence of a discrete global clock, even though processes do not have access to it. The range of the clock's ticks is the set of natural numbers.

Mode of failure. Several modes of failure have been introduced in the literature (see [Cri91, Sch93] for brief surveys). We concentrate on two modes of failure: the *crash-stop* model and the *crash-recover* model. In the crash-stop model, once a process has crashed, it never recovers. If a process p is able to execute requests at a certain time τ (i.e., p did not fail until time τ) we say that p is *up* at time τ. Otherwise, we say that p is *down* at time τ. A process that never crashes (i.e., it is always up) is *correct*, and a process that is not correct is *faulty*.

In the crash-recover model, a process p is classified according to its behaviour concerning failures as *always-up* if p never crashes, *eventually-up* if p crashes at least once, but there is a time after which p is permanently up, *eventually-down* if there is a time after which p is permanently down, and *unstable* if p crashes and recovers infinitely many times [OGS97, ACT98]. Process p is *good* if it is either always-up or eventually-up, and *bad* if it is *eventually-down* or *unstable*. Both models of failure rule out faulty processes that execute arbitrary actions (i.e., no Byzantine faults). We further assume that processes fail independently.

Process state. There are two ways of modelling processes that crash and recover according to what happens to their local state after recovering from a crash: processes can either (1) forget the state they had before the crash (i.e., processes only have *volatile memory*), or (2) remember the state they had, or a part of it, before the crash (i.e., processes have *stable storage*). Even if a process has stable storage, it is wise to use it sparingly since accessing stable storage is more expensive than accessing volatile memory.

Process set. We distinguish between a *static* set of processes, and a *dynamic* set of processes. A static set is composed of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$, and this configuration never changes throughout the execution. Conversely, if a system has a dynamic set of processes, then at two different times during the execution, the system may be composed of distinct sets of processes. Several events may trigger a change in the current set of processes if this set is dynamic (e.g., a new process *joins* the processes that are part of the current set).

Process models in perspective. Process models proposed in the literature can be defined by combining the parameters presented above (see Table 2.1).

Model M_1 has been considered by several authors [FLP85, CT96, Sch97]. The strongest argument in favour of model M_1 is that it provides a relatively simple framework to rigorously study distributed algorithms. However, in practical scenarios it lacks flexibility since once a process has crashed, it is not allowed to recover

Model	Model criteria		
	Mode of failure	Process state	Process set
M_1	crash-stop	—	static
M_2	crash-stop	—	dynamic
M_3	crash-recover	volatile	static
M_4	crash-recover	stable	static

Table 2.1: Process models

or be replaced by another process. This drawback is one way or another overcome by the other models. Model M_2 was introduced by Isis [BJ87].[1] It does not permit processes to recover but it allows a process that has been excluded from a view to join the other processes with a new identification, which is a way round the problem encountered in model M_1. In Isis, when a new process joins a view, it receives the *state* from processes in this view (e.g., the messages that processes have received in the view). Only recent proposals have considered the crash-recover model with asynchronous processes. Model M_3 has been considered in [ACT98], and model M_4 in [OGS97, HMR97, ACT98].

2.1.2 Communication Channel Model

Communication channels can be characterised according to timing, reliability, and ordering properties. Before going into detail on each one of these properties, we define $send(m)$ and $receive(m)$ as the primitives processes use to communicate. Message m is taken from a set $\mathcal{M}$ to which all messages belong. When a process p invokes "send" with a message m as a parameter, we say that p *sends* m, and when a process q returns from the execution of "receive" with a message m as a parameter, we say that q *receives* m.

Timing properties. Timing properties are related to guarantees on transmission delays of messages, which can be bounded or unbounded. This work assumes that communication delays are unbounded.

Reliability properties. Two characterisations of communication channels according to reliability properties are *Reliable Channels* [BCBT96] and *Quasi-Reliable channels* [ACT97]. Reliable Channels satisfy the following properties:

(No creation) If process q receives message m from p, then p sends m to q.

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

(No loss) If p sends m to q, and q is correct, then q eventually receives m.

[1] In Isis, process sets are called *views*.

Quasi-Reliable Channels are specified by replacing the No Loss property of Reliable Channels by the following property:

> (QUASI-NO LOSS) If p sends m to q, and p and q are correct, then q eventually receives m.

Quasi-Reliable Channels define weaker constraints than Reliable Channels, that is, any execution that satisfies Reliable Channels properties, also satisfies Quasi-Reliable properties, however, the contrary is not true. Figure 2.1 shows an execution involving processes p and q that satisfies the quasi-no loss property, but does not satisfy the no loss property. In Figure 2.1, message m is never received by process q.

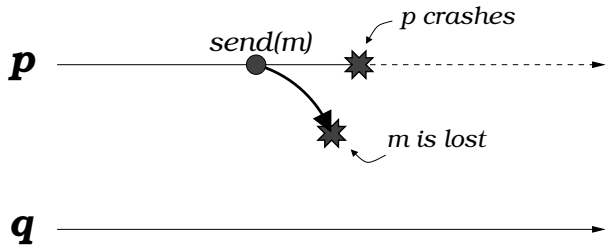

Figure 2.1: Quasi-Reliable Channels

Ordering properties. Reliable and Quasi-Reliable Channels guarantees can be augmented with ordering properties. Two particular ordering properties are *FIFO order*, and *causal order*. FIFO order is defined as shown next.

> (FIFO ORDER) If p sends m to q before sending m' to q, then q does not receive m' before m.

Causal order is defined based on Lamport's happened before relation $\rightarrow$ [Lam78]. Let a, b, and c be events in a distributed system. The relation $a \rightarrow b$ (i.e., a happens before b) holds if and only if (1) a and b are events in the same process and a occurred before b, or (2) a is the event of sending a message m in a process and b is the event of receipt of message m in another process, or (3) there exists an event c such that $a \rightarrow c$, and $c \rightarrow b$. Causal order is defined as follows.

> (CAUSAL ORDER) If m and m' are two messages received by some process p, and $send(m) \rightarrow send(m')$, then $receive(m) \rightarrow receive(m')$ in p.

Figures 2.2 and 2.3 depict FIFO and Causal Channels. Causal Channels are stronger than FIFO Channels, that is, Causal Channels preserve FIFO order. The executions in Figures 2.2 and 2.3 satisfy both Reliable Channels and Quasi-Reliable specifications.

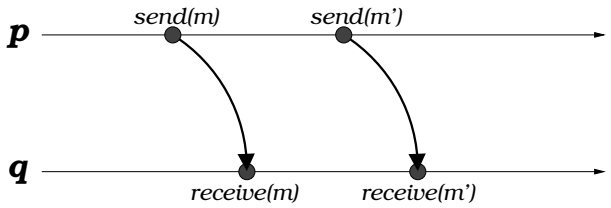

Figure 2.2: FIFO Channels

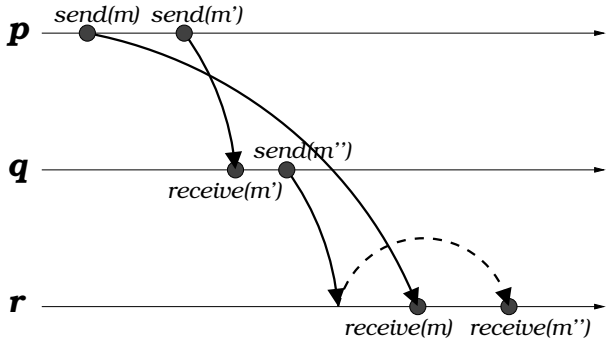

Figure 2.3: Causal Channels

2.1.3 Asynchronous Systems

Asynchronous systems are modelled by asynchronous processes that communicate through channels with unbounded transmission delays. Asynchronous systems define a very general model, and several impossibility results have been based on them. In [FLP85], it has been shown that Consensus (see Section 2.2) is not solvable in asynchronous systems subject to crash-stop failures (Model M_1). Impossibility results have also been presented for asynchronous systems with a dynamic set of processes [CHTCB96] (model M_2), and asynchronous systems with processes that can crash and recover [ACT98] (models M_3 and M_4). The latter result defines minimal bounds for solving Consensus when processes' state is volatile and stable.

2.1.4 Failure Detectors

To circumvent the *Fischer-Lynch-Paterson impossibility* result [FLP85] (FLP for short), asynchronous systems with a static set of crash-stop processes have been augmented with failure detectors [CT96]. Each process p in Π has access to a local failure detector module $\mathcal{D}_p$ that provides (possibly incorrect) information about the processes that are suspected to have crashed. A failure detector may make mistakes, that is, (1) it may suspect a process that has not failed or (2) never suspect a process that has failed. Failure detectors have been classified according to *accuracy* and *completeness* properties which characterise the mistakes they can make [CT96].

COMPLETENESS. There are two completeness properties.

▷ *Strong Completeness:* eventually every process that crashes is permanently suspected by every correct process.

▷ *Weak Completeness:* eventually every process that crashes is permanently suspected by some correct process.

ACCURACY. There are four accuracy properties.

▷ *Strong Accuracy:* no process is suspected before it crashes.

▷ *Weak Accuracy:* some correct process is never suspected.

▷ *Eventual Strong Accuracy:* there is a time after which correct processes are not suspected by any correct process.

▷ *Eventual Weak Accuracy:* there is a time after which some correct process is never suspected by any correct process.

Table 2.2 summarises all classes of failure detectors. Throughout this work, we do not consider any failure detector in particular, nevertheless, we assume that the system is augmented with failure detectors so that Consensus can be solved. Moreover, Chapters 4 and 5 need a failure detector that guarantees Strong Completeness (i.e., Accuracy is not relevant).

	Accuracy			
Completeness	Strong	Weak	Eventually Strong	Eventually Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

Table 2.2: Failure detectors classes

It has been shown in [CHT96] that $\diamond\mathcal{W}$ is the weakest failure detector to solve Consensus in asynchronous systems subject to crash-stop failures, and [CT96] shows that any given failure detector $\mathcal{D}$ that satisfies weak completeness can be reduced into a failure detector $\mathcal{D}'$ that satisfies strong completeness, that is, $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ are equivalent.

The works presented in [ACT98, OGS97] redefine failure detectors in asynchronous systems where processes can crash and recover. In this thesis, we concentrate on failure detectors in the crash-stop model.

2.1.5 Algorithms, Runs and Problems

When discussing distributed protocols, it is important to characterise the notions of algorithm, run, and problem. In the following, we provide definitions for these terms in the context of asynchronous processes in the crash-stop model, which is the

model considered in Chapters 3 and 4. Chapter 2 is based on a formalism specific to databases, introduced in Section 2.3.

An *algorithm* A is a collection of n deterministic automata, one per process, and computation proceeds in *steps* of A. In the crash-stop model, in each step, a process atomically (1) receives a (possibly empty) message that was sent to it, (2) queries its failure detector module, (3) modifies its state, and (4) may send a message to a single process [CT96].

Informally, a *run* R of A defines a (possibly infinite) sequence of steps of A. There is a close relation between system models, algorithms, and problems, in that a system model M determines the set of runs that an algorithm A can produce in M, and a *problem* specification P (or simply a problem) is defined as requirements on sets of runs.

2.2 Fault-Tolerant Broadcasts and Related Problems

In this section, we define Reliable Broadcast, Atomic Broadcast, Consensus, and Non-Blocking Atomic Commitment in asynchronous systems with processes that crash and stop. The Non-Blocking Atomic Commitment definition further assumes that the system is augmented with failure detectors. The abstractions presented in this section lay the basis for the work developed in Chapters 3, 4, and 5.

2.2.1 Reliable Broadcast

Reliable Broadcast is defined by the primitives *R-broadcast*(m) and *R-deliver*(m), which satisfy the following properties [HT93].

> (Validity) If a correct process R-broadcasts a message m, then it eventually R-delivers m.
>
> (Agreement) If a correct process R-delivers a message m, then all correct processes eventually R-deliver m.
>
> (Uniform integrity) For every message m, every process R-delivers m at most once, and only if m was previously R-broadcast by *sender*(m).

R-broadcast and R-deliver may be build over Quasi-Reliable Channels, which offer weaker guarantees than Reliable Broadcast (see Figure 2.4). In the crash-stop model, Reliable Broadcast can be solved by the following algorithm, resilient to $n-1$ process crashes [CT96]. Whenever a process p wants to R-broadcast a message m, p sends m to all processes. Once a process q receives m, if $q \neq p$ then q sends m to all processes, and, in any case, q R-delivers m (see the Appendix for a detailed presentation of this algorithm).

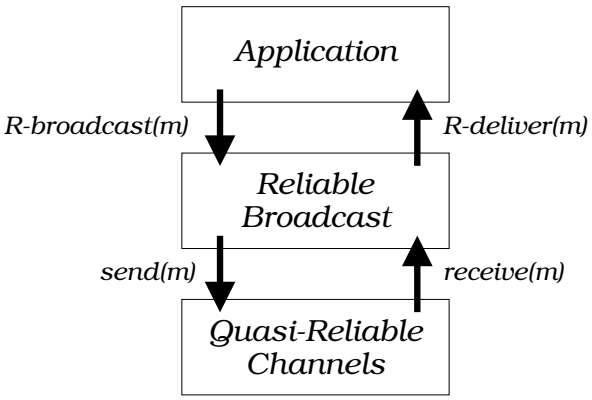

Figure 2.4: Communication abstraction

2.2.2 Atomic Broadcast

Atomic Broadcast is defined by the primitives *A-broadcast*(m) and *A-deliver*(m). In addition to the properties of Reliable Broadcast, Atomic Broadcast satisfies the *total order* property [HT93].

> (TOTAL ORDER) If two correct processes p and q A-deliver two messages m and m', then p A-delivers m before m' if and only if q A-delivers m before m'.

The total order induced on the A-deliver is represented by the relation $\prec$. Thus, if message m is A-delivered before message m', then *A-deliver*(m) $\prec$ *A-deliver*(m').

Stronger definitions of Reliable and Atomic Broadcast can be obtained by augmenting the properties previously presented with FIFO and Causal Order constraints. The resulting definitions are FIFO Broadcast (FIFO Order + Reliable Broadcast), Causal Broadcast (Causal Order + Reliable Broadcast), FIFO Atomic Broadcast (FIFO Order + Atomic Broadcast), and Causal Atomic Broadcast (Causal Order + Atomic Broadcast). Figure 2.5 depicts the relationship among broadcast primitives [HT93].

2.2.3 Consensus

Consensus is defined by the primitives *propose*(v), and *decide*(v), which satisfy the following properties.

> (TERMINATION) Every correct process eventually decides some value.
>
> (UNIFORM INTEGRITY) Every process decides at most once.
>
> (AGREEMENT) No two correct processes decide differently.
>
> (UNIFORM VALIDITY) If a process decides v, then v was proposed by some process.

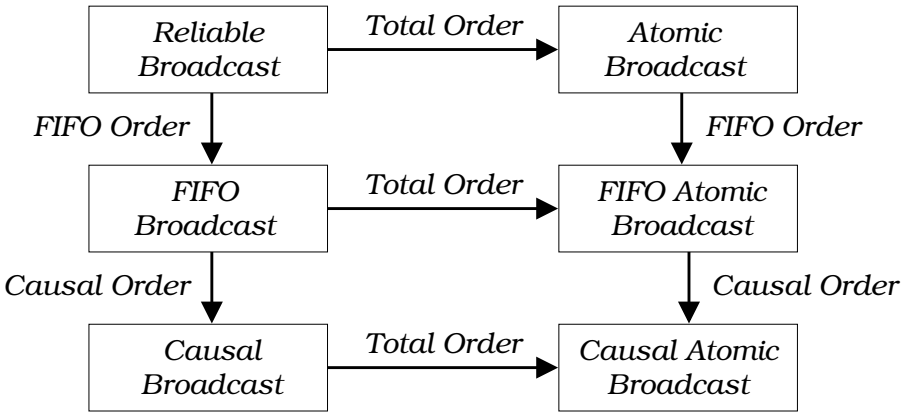

Figure 2.5: Relationship among broadcast primitives

Consensus can be solved in crash-stop asynchronous systems augmented with failure detectors. In [CT96] the authors present two algorithms that solve Consensus. One uses a failure detector of class $\mathcal{S}$ and tolerates $f < n$ failures, and the other uses a failure detector of class $\diamond\mathcal{S}$ and tolerates $f < n/2$ failures. Another algorithm that solves Consensus in the crash-stop model using a failure detector of class $\diamond\mathcal{S}$ is the Early Consensus algorithm [Sch97]. The Early Consensus algorithm tolerates $f < n/2$ failures and to a certain extend, is more efficient than the algorithm based on $\diamond\mathcal{S}$ proposed in [CT96]. The Consensus algorithm presented in [CT96] using a failure detector of class $\diamond\mathcal{S}$, and the Early Consensus algorithm are presented in the Appendix.

Consensus and Atomic Broadcast have been shown in the literature to be equivalent in the crash-stop model [CT96]. The equivalence result basically states that Atomic Broadcast can be reduced to Consensus (see the Appendix), and Consensus can be reduced to Atomic Broadcast. The Consensus to Atomic Broadcast reduction consists in having *propose*(v) execute *A-broadcast*(v), and *decide*(v) occurring after the first *A-deliver*(v).

2.2.4 Non-Blocking Atomic Commitment

Non-Blocking Atomic Commitment is defined by the primitives *AC-vote*(v) and *AC-decide*(v), $v \in \{commit, abort\}$, which ensure the following properties.

(Uniform Agreement) No two participants AC-decide differently.

(Uniform Validity) If a process AC-decides *commit*, then all processes have voted *commit*.

(Termination) Every correct process eventually AC-decides.

(Non-Triviality) If all processes vote *commit*, and there is no failure, then every correct process eventually AC-decides *commit*.

Non-Blocking Atomic Commitment has been shown to be unsolvable in asynchronous systems subject to crash failures, even if augmented with failure detectors of the class $\diamond\mathcal{P}$ or $\mathcal{S}$ [Gue95]. However, a weaker version of Atomic Commit (Non-Blocking Weak Atomic Commit) can be reducible to Consensus. Non-Blocking Weak Atomic Commit replaces the previously defined Non-Triviality property by the following.

> (WEAK NON-TRIVIALITY) If all processes vote *commit*, and no process is ever suspected, then every correct process eventually AC-decides *commit*.

2.3 Database Definitions

In this section, we formally define transactions and histories, present the ACID transaction properties, and discuss model assumptions usually associated with databases. Formal definitions of transactions and histories will be useful to prove replicated databases protocols correct.

2.3.1 Transactions and Histories

Informally, a transaction is a set of database operations that finishes with a Commit or an Abort operation. Let $\Gamma = \{x_1, x_2, \ldots, x_m\}$ be a database, and $r[x_k]$ and $w[x_k]$ be a read and a write operation on data item x_k, $x_k \in \Gamma$, respectively, and c and a be the commit and abort operations. Formally, transaction t_i is defined as a partial order on read and write operations with ordering relation $<_i$, where

1. $t_i \subseteq \{r_i[x_k], w_i[x_k] : x \in \Gamma\} \cup \{a_i, c_i\}$;
2. $a_i \in t_i$ iff $c_i \notin t_i$;
3. let o be c_i or a_i (whichever is in t_i), for any other $o' \in t_i$, $o' <_i o$; and
4. for any two operations $r_i[x_k]$ and $w_i[x_k]$ such that $r_i[x_k], w_i[x_k] \in t_i$, then either $r_i[x_k] <_i w_i[x_k]$ or $w_i[x_k] <_i r_i[x_k]$.

Transactions executing in a database are formalised by histories [BHG87]. Let $T = \{t_1, t_2, \ldots, t_j\}$ be a set of transactions. A *complete history* H over T is a partial order on read and write operations with ordering relation $<_H$ where

1. $H = \cup_{i=1}^{j} t_i$;
2. $\cup_{i=1}^{j} <_i \subseteq <_H$; and
3. for any two operations $w[x_k]$ and $o[x_k], o \in \{r, w\}$, issued by different transactions in H, either $w[x_k] <_H o[x_k]$ or $o[x_k] <_H w[x_k]$.

A *history* is a prefix of a complete history. Given some history H, the *committed projection of* H, denoted $C(H)$, is the history obtained from H, by eliminating all operations that do not belong to transactions committed in H.

2.3.2 Transaction Properties

Transactions satisfy the ACID properties. The ACID acronym stands for Atomicity, Consistency, Isolation, and Durability. The ACID properties are defined as follows [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 the 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 other transactions executed either before t or after t, but not both.
>
> (Durability) Once a transaction completes successfully (commits), its changes to the state survive failures.

From the viewpoint of the history definition presented in the previous section, the atomicity property states that the study of the correctness of database protocols (i.e., serialisability), should concentrate on the committed projections of the histories produced by these protocols.

Consistency is not relevant in the history formalism previously defined, since it deals with semantic meaning about the transformations performed by transactions on the database, and the history formalism is not strong enough to capture this abstraction level.

Isolation has received a lot of attention by database researchers, mainly in the early 70's. According to the transaction and history formalism presented in the previous section, a database protocol ensures isolation if the committed projection of any history it produces does not have cycles [BHG87]. Isolation is also known as serialisability, or, in the context of replicated databases, one-copy serialisability.

The durability property is highly dependent on the assumptions made about processes. For example, most database systems consider that database sites (or processes) always recover after a crash, and have access to stable storage. In this scenario, durability can be enforced by carefully storing critical information in stable storage [Had88].

2.4 Discussion

A model is a simplification of a real system, allowing to study it in depth, without having to worry about details. Usually, the more complex the model, the closer to the reality it is, however, complex models make the approach to the problem difficult.

In this chapter, we have characterised distributed systems and database systems by the models usually presented in the literature.

The thesis focuses on two distinct system models. In Chapter 3, we consider that processes crash and recover, have access to stable storage, and belong to a static set of processes. In Chapters 4 and 5 we consider that processes crash and stop, do not have stable storage, and belong to a static set of processes. In Chapters 4 and 5, processes are fully connected by Quasi-Reliable Channels, and Chapter 4 further assumes that communication channels are FIFO.

Thus, the two models considered in this thesis differ on the mode of failure of processes. This distinction has simplified the work in Chapters 4 and 5. However, the intuitions behind the ideas proposed in Chapters 4 and 5 do not depend on details about the crash-stop model, and can be extended to the crash-recover model.

Chapter 3

The Database State Machine

> First things first, but not necessarily in that order.
>
> **Doctor Who**

This chapter introduces the first contribution of the thesis, the Database State Machine. The Database State Machine is, from the user's point of view, a high-performance and high-availability database that offers strong consistency (i.e., serialisability). From the system's viewpoint, the Database State Machine is a mechanism to handle replication in a cluster of workstations connected by a standard communication network.

From the model perspective presented in the previous chapter, the Database State Machine considers a static set of processes that have access to stable storage. Processes communicate through an Atomic Broadcast primitive.

Compared to other database approaches that also provide high-availability, the Database State Machine does not sacrifice performance (it minimises inter site synchronisation and eliminates distributed deadlocks), nor data consistency. Furthermore, by relying on a cluster of workstations connected by a standard communication network, the Database State Machine does not depend on specialised hardware.

In this chapter, we recall the deferred update replication technique and the principle of the state machine approach [Sch90], which define the general framework for the Database State Machine, and present the architecture of the Database State Machine and the Reordering technique. The performance of the Database State Machine is analysed with simulation and probabilistic models.

3.1 Deferred Update Replication Framework

Before presenting the Database State Machine approach, we describe the deferred update replication principle in detail and introduce some additional notation. We also

provide a general algorithm that will lay the basis for the Database State Machine algorithm.

3.1.1 Deferred Update Replication Principle

In the deferred update replication technique, transactions are locally executed at one database site, and during their execution, no interaction between other database sites occurs (see Figure 3.1). Transactions are locally synchronised at database sites according to some concurrency control mechanism [BHG87]. Hereafter, we assume that the concurrency control mechanism used by every database site to local synchronise transactions is the strict two phase locking rule. When a client requests the transaction commit, the transaction's updates (e.g., the redo log records) and some control structures are propagated to all database sites, where the transaction will be certified and, if possible, committed. This procedure, starting with the commit request, is called *termination protocol*. The objective of the termination protocol is twofold: (i) propagating transactions to database sites, and (ii) certifying them.

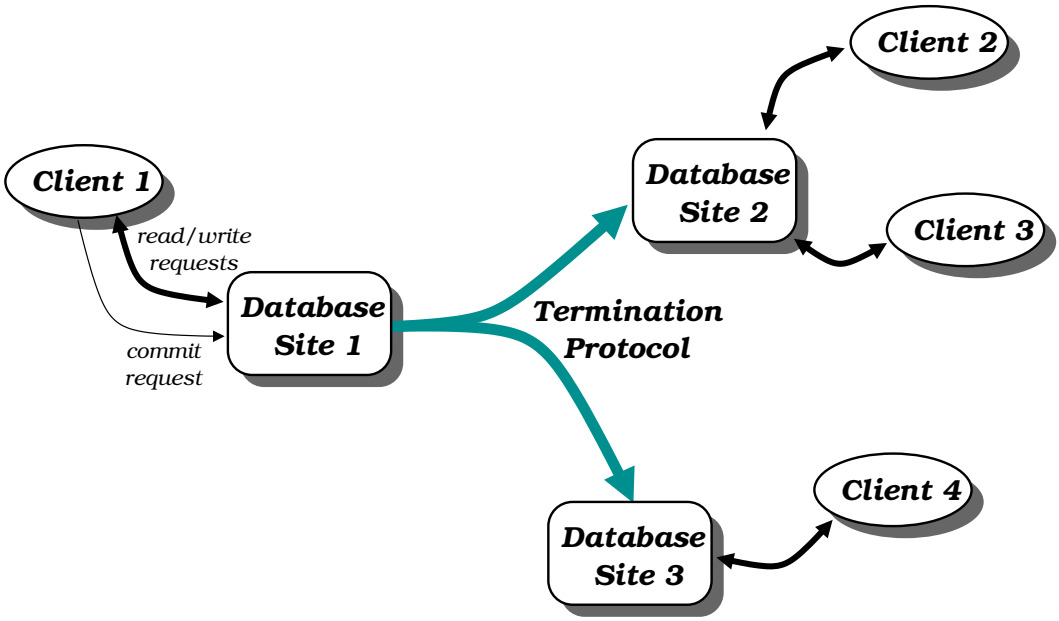

Figure 3.1: Deferred update technique

The certification test aims at ensuring one-copy serialisability. It decides to abort a transaction if the transaction's commit would lead the database to an inconsistent state (i.e., non-serialisable). For example, consider two concurrent transactions, t_a and t_b, that are executed at different database sites, and that update a common data item. When t_a and t_b request the commit, the certification test has to realise whether consistency may be violated and, if this is the case, sort out the problem by deciding to abort one or both transactions (e.g., if there is no guarantee that t_a and t_b arrive at all sites in the same order, both transactions have to be aborted [AAS97], however, if the certifier knows that t_a is received before t_b at all sites, or the other

way round, then just t_a, respectively t_b, has to be aborted [PGS98]).

3.1.2 Transaction States

During its processing, a transaction passes through some well-defined states (see Figure 3.2). The transaction starts in the *executing state*, when its read and write operations are locally executed at the database site where it was initiated. When the client that initiates the transaction requests the commit, the transaction passes to the *committing state* and is sent to the other database sites. A transaction received by a database site in the context of the termination protocol is also in the committing state, and it remains in the committing state until its fate is known by the database site (i.e., *commit* or *abort*). The different states of a transaction t_a at a database site s_i are denoted $Executing(t_a, s_i)$, $Committing(t_a, s_i)$, $Committed(t_a, s_i)$, and $Aborted(t_a, s_i)$. The executing and committing states are transitory states, whereas the committed and aborted states are final states.

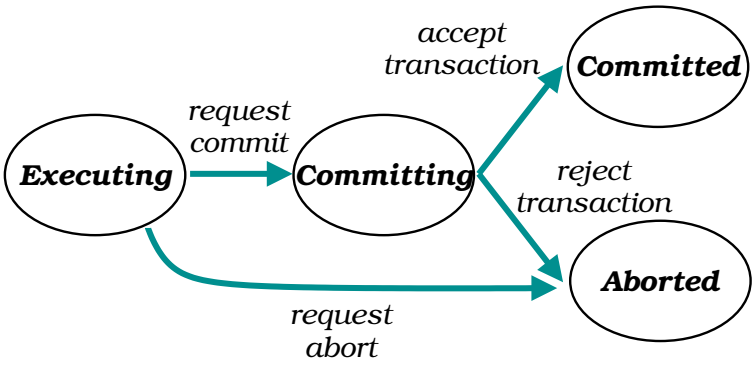

Figure 3.2: Transaction states

3.1.3 Deferred Update Replication Algorithm

We describe next a general algorithm for the deferred update replication technique. To simplify the presentation, we consider a particular client c_k that sends requests to a database site s_i in behalf of a transaction t_a.

1. Read and write operations requested by the client c_k are executed at s_i according to the strict two phase locking (strict 2PL) rule. From the start until the commit request, transaction t_a is in the executing state.

2. When c_k requests t_a's commit, t_a is immediately committed if it is a read-only transaction (nevertheless, read-only transactions may be aborted during their execution, as discussed later). If not, t_a passes to the committing state, and the database site s_i triggers the termination protocol for t_a: the updates performed by t_a, as well as its readset and writeset, are sent to all database sites.

3. Eventually, every database site s_j certifies t_a. The certification test takes into account every transaction t_b known by s_j that *conflicts* with t_a (see Section 3.1.4). It is important that all database sites reach the same decision on the final state of t_a, which may require some coordination among database sites. Such coordination can be achieved, for example, by means of an Atomic Commitment protocol, or, as it will be shown in Section 3.2, by using an Atomic Broadcast primitive.

4. If t_a is serialisable with the previous committed transactions in the system (e.g., t_a passes the certification test), all its updates will be applied to the database. Transactions in the execution state at each site s_j holding locks on the data items updated by t_a are aborted.

5. The client c_k receives the outcome for t_a from site s_i as soon as s_i can determine whether t_a will be committed or aborted. The exact moment this happens depends on how the termination protocol is implemented, and will be discussed in Section 3.2.

Queries do not execute the certification test, nevertheless, they may be aborted during their execution due to local deadlocks and by non-local committing transactions when granting their write locks (see Section 3.5). The algorithm presented above can be modified in order to reduce or completely avoid aborting read-only transactions. For example, if queries are pre-declared as so, once an update transaction passes the certification test, instead of aborting a query that holds a read lock on a data item it wants to update, the update transaction waits for the query to finish and release the lock. In this case, update transactions have the highest priority in granting write locks, but they wait for queries to finish. Read-only transactions can still be aborted due to deadlocks, though. However, multiversion data item mechanisms can prevent queries from being aborted altogether. In [SA93], read-only transactions are executed using a fixed view (or version) of the database, without interfering with the execution of update transactions.

3.1.4 Transaction Dependencies

In order for a database site s_i to certify a committing transaction t_a, s_i must be able to tell which transactions conflict with t_a up to the current time. A transaction t_b *conflicts* with t_a if t_a and t_b have *conflicting operations* and t_b does not *precede* t_a. Two operations conflict if they are issued by different transactions, access the same data item and at least one of them is a write. The precede relation between two transactions t_a and t_b is defined as follows. (a) If t_a and t_b execute at the same database site, t_b precedes t_a if t_b enters the committing state before t_a. (b) If t_a and t_b execute at different database sites, say s_i and s_j, respectively, t_b precedes t_a if t_b commits at s_i before t_a enters the committing state at s_i. Let $site(t)$ identify the database site where transaction t was executed, and $committing(t)$ and $commit(t)_{s_j}$ be the events that represent, respectively, the request for commit and the commit of t at s_j. The event $committing(t)$ only happens at the database site s_i where t was

executed, and the event $commit(t)_{s_j}$ happens at every database site s_j. We formally define that transaction t_b precedes transaction t_a, denoted $t_b \rightarrow t_a$, as

$$t_b \rightarrow t_a \equiv \begin{cases} committing(t_b) \xrightarrow{e} committing(t_a) & \text{if } site(t_a) = site(t_b), \\ commit(t_b)_{site(t_a)} \xrightarrow{e} committing(t_a) & \text{otherwise,} \end{cases}$$

where $\xrightarrow{e}$ is Lamport's order relation between system events [Lam78]. The relation $t_b \not\rightarrow t_a$ establishes that t_b does not precede t_a. If $site(t_a) = site(t_b)$, $t_b \not\rightarrow t_a$ is equivalent to $committing(t_b) \overset{e}{\not\rightarrow} committing(t_a)$. Since local events in a site are totally ordered, $committing(t_b) \overset{e}{\not\rightarrow} committing(t_a) \equiv committing(t_a) \xrightarrow{e} committing(t_b)$, and so, $t_b \not\rightarrow t_a \equiv t_a \rightarrow t_b$. If $site(t_a) \neq site(t_b)$, $t_b \not\rightarrow t_a$ is equivalent to $commit(t_b)_{site(t_a)} \overset{e}{\not\rightarrow} committing(t_a)$, or $committing(t_a) \xrightarrow{e} commit(t_b)_{site(t_a)}$.

The deferred update replication does not require any distributed locking protocol to synchronise transactions during their execution. Therefore, network bandwidth is not consumed by synchronising messages, and there are no distributed deadlocks. However, transactions may be aborted due to conflicting accesses. In the next sections, we show that the deferred update replication technique can be implemented using the state machine approach, and that this approach allows optimisations that can reduce transaction abortion due to conflicting accesses.

3.2 A Database as a State Machine

The deferred update replication technique can be implemented as a state machine. In this section, we recall the principle of the state machine approach, and discuss the details of the Database State Machine and its implications to the way transactions are processed.

3.2.1 The State Machine Approach

The state machine approach [Sch90], also called active replication, is a non-centralised replication coordination technique. Its key concept is that all replicas (or database sites) receive and process the same sequence of requests. Replica consistency is guaranteed by assuming that when provided with the same input (e.g., a client request) each replica will produce the same output (e.g., state change). This assumption implicitly implies that replicas have a deterministic behaviour.

The way requests are disseminated among replicas can be decomposed into two requirements [Sch90]:

(Agreement.) Every non-faulty replica receives every request.

(Order.) If a replica first processes request req_1 before req_2, then no replica processes request req_2 before request req_1.

The order requirement can be weakened if some semantic information about the requests is known. For example, if two requests commute, that is, independently of the order they are processed they produce the same final states and sequence of outputs, then it is not necessary that order be enforced among the replicas for these two requests.

3.2.2 The Termination Protocol

The termination protocol presented in Section 3.1 can be turned into a state machine (i.e., made deterministic) as follows. Whenever a client requests a transaction's commit, the transaction's updates, its readset and writeset (or, for short, the transaction) are atomically broadcast to all database sites. Each database site will behave as a state machine, and the agreement and order properties required by the state machine approach are ensured by the Atomic Broadcast primitive.

The database sites, upon delivering and processing the transaction, should eventually reach the same state. In order to accomplish this requirement, delivered transactions should be processed with certain care. Before delving deeper into details, we describe the database modules involved in the transaction processing. Figure 3.3 abstractly presents such modules and the way they are related to each other.[1] Transaction execution, as described in Section 3.1, is handled by the *Transaction Manager*, the *Lock Manager*, and the *Data Manager*. The *Certifier* executes the certification test for an incoming transaction. It receives the transactions delivered by the *Atomic Broadcast* module. On certifying a transaction, the Certifier may ask information to the data manager about already committed transactions (e.g., logged data). If the transaction is successfully certified, its write operations are transmitted to the Lock Manager, and once the write locks are granted, the updates can be performed.

To make sure that each database site will converge to the same state after processing committing transactions, each certifier has to (1) reach the same decision when certifying transactions, and (2) guarantee that write-conflicting transactions are applied to the database in the same order (since transactions whose writes do not conflict are commutable). The first constraint is ensured by providing each certifier with the same set of transactions and using a deterministic certification test. The second constraint can be attained if the certifier ensures that write-conflicting transactions grant their locks in the same order that they are delivered. This requirement is straightforward to implement, nevertheless, it reduces concurrency in the certifier.

3.2.3 The Termination Algorithm

The procedure executed on delivering the request of a committing update transaction t_a is detailed next. For the discussion that follows, the *readset* $RS(t_a)$ and the *writeset* $WS(t_a)$ are sets containing the identifiers of the data items read and written by t_a,

[1] In a database implementation, these distinctions may be much less apparent, and the modules more tightly integrated [GR93]. However, for presentation clarity, we have chosen to separate the modules.

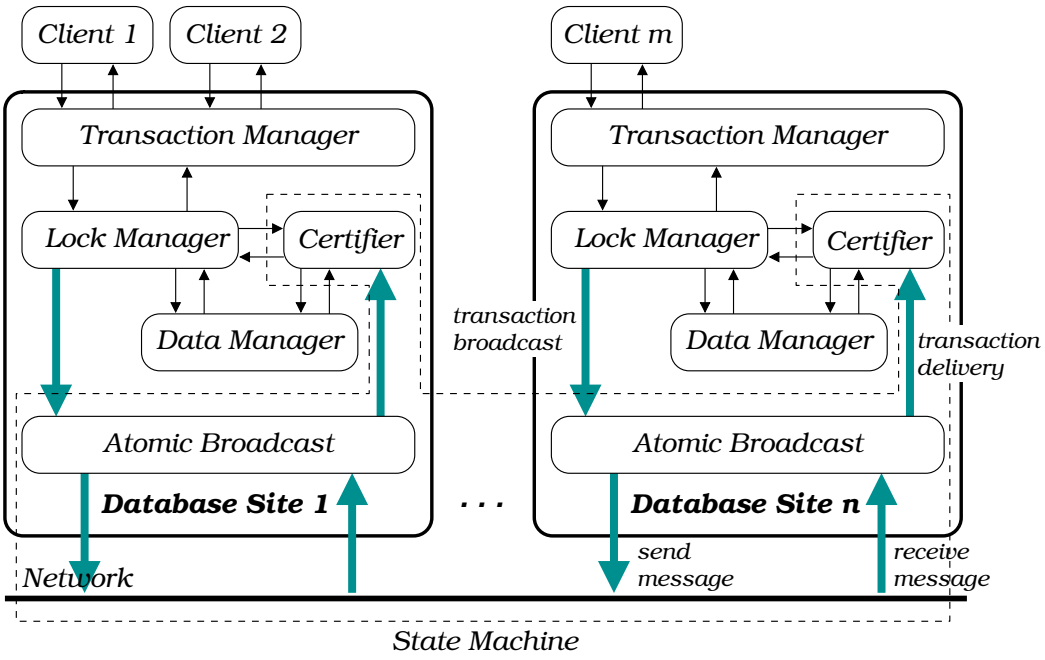

Figure 3.3: Termination protocol based on Atomic Broadcast

respectively, during t_a's execution. Assuming that t_a was executed at database site s_i, every database site s_j, after delivering t_a, performs the following steps:

1. *Certification test.* Database site s_j commits t_a (i.e., t_a passes from the committing state to the committed state at s_j) if there is no committed transaction t_b at s_j that conflicts with t_a. The notion of conflicting operations defined in Section 3.1.4 is weakened, and just write operations performed by committed transactions and read operations performed by t_a are considered (i.e., write-read conflicts). Read-write conflicts are not relevant since only committed transactions take part in t_a's certification test, and write-write conflicts are solved by guaranteeing that all t_a's updates are applied to the database after all the updates performed by committed transactions (up to the current time).

   The certification test is formalised next as a condition for a state transition from the committing state to the committed state (see Figure 3.2):

$$Committing(t_a, s_j) \rightsquigarrow Committed(t_a, s_j) \equiv \left[\begin{array}{l} \forall t_b,\ Committed(t_b, s_j) : \\ \\ t_b \rightarrow t_a \vee (WS(t_b) \cap RS(t_a) = \emptyset) \end{array} \right]$$

   The condition for a transition from the committing state to the aborted state is the complement of the right side of this expression.

   Once t_a has been certified by database site s_i, where it was executed, s_i can inform t_a's outcome to the client that requested t_a's execution.

2. *Commitment.* If t_a is not aborted, it passes to the commit state, the locks for the data items it has written are requested, and once granted, t_a's updates are performed. There are three cases to consider on granting the write locks requested by t_a.

   (a) *There is a transaction t_b in execution at s_j whose read or write locks conflict with t_a's writes.* In this case t_b is aborted by s_j, and therefore, all t_b's read and write locks are released.

   (b) *There is a transaction t_b, that was executed locally at s_j and requested the commit, but has not been A-delivered yet at s_j.* Since t_b executed locally at s_j, t_b has its write locks on the data items it updated. If t_b commits, its writes will overwrite t_a's (i.e., the ones that overlap) and, in this case, t_a need neither request these write locks nor process the updates over the database. This is similar to Thomas' Write Rule [Tho79]. However, if t_b is later aborted (i.e., it does not pass the certification test), the database should be restored to a state without t_b, for example, by applying t_a's redo log entries to the database.

   (c) *There is a transaction t_b that has passed the certification test and has granted its write locks at s_j, but it has not released them yet.* In this case, t_a waits for t_b to finish its updates and release its write locks.

An important aspect of the termination algorithm presented above is that the Atomic Broadcast is the only form of interaction between database sites. The Atomic Broadcast properties guarantee that every database site will certify a transaction t_a using the same set of committed transactions. It remains to be shown how each database site builds such a set. If transactions t_a and t_b execute at the same database site, this can be evaluated by identifying transactions that execute at the same site (e.g., each transaction carries the identity of the site where it was initiated) and associating local timestamps to the committing events of transactions.

If t_a and t_b executed at different sites, this is done as follows. Every transaction commit event is timestamped with the order the transaction was A-delivered. The Atomic Broadcast properties ensure that each database site associates the same timestamps to the same transactions, and there are no two transactions with the same timestamp. Each transaction t has a $committing(t)$ field that stores the commit timestamp of the last locally committed transaction when t passes to the committing state (see Figure 3.4). The $committing(t)$ field is broadcast to all database sites together with t. When a database site s_j certifies t_a, all committed transactions that have been delivered by s_j with commit timestamp greater than $committing(t_a)$ take part in the set of committed transactions used to certify t_a ($t_{(2)}$ to $t_{(m)}$ in Figure 3.4). Such a set of committed transactions only contains transactions that do not precede t_a.

3.2.4 Algorithm Correctness

The Database State Machine algorithm is proved correct using the multiversion formalism of [BHG87]. Although we do not explicitly use multiversion databases, our

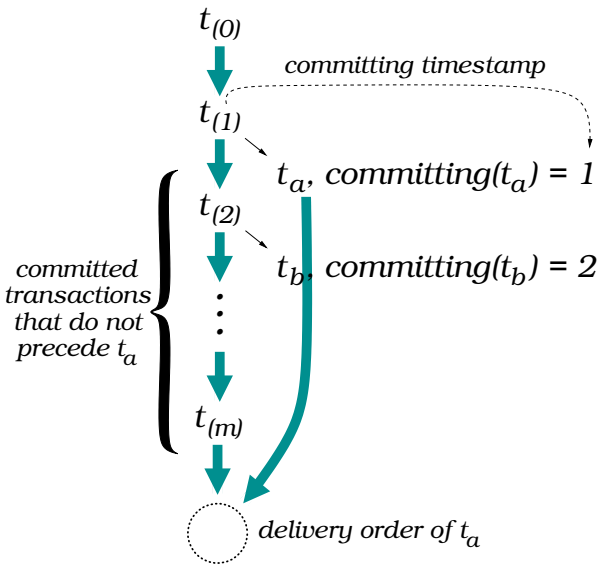

Figure 3.4: Transaction precedences

approach can be seen as so, since replicas of a data item located at different database sites can be considered as different versions of this data item [BHG87].

We first define $C(H)_{s_i}$ as a multiversion history derived from the system history H, just containing operations of committed transactions involving data items stored at s_i. We denote $w_a[x_a]$ a write by t_a (as writes generate new data versions, the subscript in x for data writes is always the same as the one in t) and $r_a[x_b]$ a read by t_a of data item x_b.

The multiversion formalism uses a multiversion serialisation graph ($MVSG(C(H)_{s_i})$ or $MVSG_{s_i}$ for short) and consists in showing that all the histories produced by the algorithm have a multiversion serialisation graph that is acyclic [BHG87]. We denote $MVSG^k_{s_i}$ a particular state of the multiversion serialisation graph for database site s_i. The multiversion serialisation graph passes from one state $MVSG^k_{s_i}$ into another $MVSG^{k+1}_{s_i}$ when a transaction is committed at s_i.

We exploit the state machine characteristics to structure our proof in two parts. In the first part, Lemma 3.1 shows that, by the properties of the Atomic Broadcast primitive and the determinism of the certifier, every database site $s_i \in \Sigma_D$ eventually constructs the same $MVSG^k_{s_i}$, $k \geq 0$. In the second part, Lemmas 3.2 and 3.3 show that every $MVSG^k_{s_i}$ is acyclic.

Lemma 3.1 *If a database site s_i constructs a multiversion serialisation graph $MVSG^k_{s_i}, k \geq 0$, then every database site s_j eventually constructs the same multiversion serialisation graph $MVSG^k_{s_j}$.*

Proof: The proof is by induction. (Base step.) When the database is initialised, every database site s_j has the same empty multiversion serialisation graph

$MVSG^0_{s_j}$. (INDUCTIVE STEP - ASSUMPTION.) Assume that every database site s_j that has constructed a multiversion serialisation graph $MVSG^k_{s_j}$ has constructed the same $MVSG^k_{s_j}$. (INDUCTIVE STEP - CONCLUSION.) Consider t_a the transaction whose committing generates, from $MVSG^k_{s_j}$, a new multiversion serialisation graph $MVSG^{k+1}_{s_j}$. In order to do so, database site s_j must deliver, certify and commit transaction t_a. By the order property of the Atomic Broadcast primitive, every database site s_j that delivers a transaction after installing $MVSG^k_{s_j}$, delivers t_a, and, by the atomicity property, if one database site delivers transaction t_a, then every database site delivers t_a. To certify t_a, s_j takes into account the transactions that it has already locally committed (i.e., the transactions in $MVSG^k_{s_j}$). Thus, based on the same local state ($MVSG^k_{s_j}$), the same input (t_a), and the same (deterministic) certification algorithm, every database site eventually constructs the same $MVSG^{k+1}_{s_j}$. □

We show next that every history $C(H)_{s_i}$ produced by a database site s_i has an acyclic $MVSG_{s_i}$ and, therefore, is $1SR$ [BHG87]. Given a multiversion history $C(H)_{s_i}$ and a version order $\ll$, the multiversion serialisation graph for $C(H)_{s_i}$ and $\ll$, $MVSG_{s_i}$, is a serialisation graph with read-from and version order edges. A read-from relation $t_a \hookrightarrow t_b$ is defined by an operation $r_b[x_a]$. There are two cases where a version-order relation $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$: (a) for each $r_c[x_b]$, $w_b[x_b]$ and $w_a[x_a]$ in $C(H)_{s_i}$ (a, b, and c are distinct) and $x_a \ll x_b$, and (b) for each $r_a[x_c]$, $w_c[x_c]$ and $w_b[x_b]$ in $C(H)_{s_i}$ and $x_c \ll x_b$. The version order is defined by the delivery order of the transactions. Formally, a version order can be expressed as follows: $x_a \ll x_b$ iff $deliver(t_a) \prec deliver(t_b)$ and $t_a, t_b \in MVSG_{s_i}$.

To prove that $C(H)_{s_i}$ has an acyclic multiversion serialisation graph ($MVSG_{s_i}$) we show that the read-from and version-order relations in $MVSG_{s_i}$ follow the order of delivery of the committed transactions in $C(H)_{s_i}$. That is, if $t_a \hookrightarrow t_b \in MVSG_{s_i}$ then $deliver(t_a) \prec deliver(t_b)$.

Lemma 3.2 *If there is a* read-from *relation* $t_a \hookrightarrow t_b \in MVSG_{s_i}$ *then* $deliver(t_a) \prec deliver(t_b)$.

PROOF: A read-from relation $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$ if $r_b[x_a] \in C(H)_{s_i}, a \neq b$. For a contradiction, assume that $deliver(t_b) \prec deliver(t_a)$. If t_a and t_b were executed at different database sites, by the time t_b was executed, t_a had not been committed at $site(t_b)$, and thus, t_b could not have read a value updated by t_a. If t_a and t_b were executed at the same database site, t_b must have read uncommitted data from t_a, since t_a had not been committed yet. However, this contradicts the strict two phase locking rule. □

Lemma 3.3 *If there is a* version-order *relation* $t_a \hookrightarrow t_b \in MVSG_{s_i}$ *then* $deliver(t_a) \prec deliver(t_b)$.

PROOF: According to the definition of version-order edges, there are two cases to consider. (1) Let $r_c[x_b]$, $w_b[x_b]$ and $w_a[x_a]$ be in $C(H)_{s_i}$ (a, b and c distinct), and

$x_a \ll x_b$, which implies $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$. It follows from the definition of version-order that $deliver(t_a) \prec deliver(t_b)$. (2) Let $r_a[x_c]$, $w_c[x_c]$ and $w_b[x_b]$ be in $C(H)_{s_i}$, and $x_c \ll x_b$, which implies $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$, and we have to show that $deliver(t_a) \prec deliver(t_b)$. For a contradiction, assume that $deliver(t_b) \prec deliver(t_a)$. From the certification test, when t_a is certified, either $t_b \rightarrow t_a$ or $WS(t_b) \cap RS(t_a) = \emptyset$. But since $x \in RS(t_a)$, and $x \in WS(t_b)$, it must be that $t_b \rightarrow t_a$.

Assume that t_a and t_b were executed at the same database site. By the definition of precedence (Section 3.1.4), t_b requested the commit before t_a (that is, $committing(t_b) \stackrel{e}{\rightarrow} committing(t_a)$). However, t_a reads x_c from t_c, and this can only happen if t_b updates x before t_c, that is, $x_b \ll x_c$, contradicting that $x_c \ll x_b$. A similar argument follows for the case where t_a and t_b were executed at distinct database sites, and we conclude that if there is a version-order relation $t_a \hookrightarrow t_b$ in $MVSG_{s_i}$ then $deliver(t_a) \prec deliver(t_b)$. □

Theorem 3.1 *Every history H produced by the Database State Machine algorithm is 1SR.*

Proof: By Lemmas 3.2 and 3.3, every database site s_i produces a serialisation graph $MVSG^k_{s_i}$ such that every edge $t_a \hookrightarrow t_b \in MVSG^k_{s_i}$ satisfies the relation $deliver(t_a) \prec deliver(t_b)$. Hence, every database site s_i produces an acyclic multiversion serialisation graph $MVSG^k_{s_i}$. By Lemma 3.1, every database site s_i constructs the same $MVSG^k_{s_i}$. By the Multiversion Serialisation Graph theorem of [BHG87], every history produced by the Database State Machine algorithm is 1SR. □

3.2.5 Coping with Unilateral Aborts

Once a transaction t is delivered and successfully certified at some database site s_i, t has to be committed at s_i. Nevertheless, it can happen that for some "local reason" (e.g., disk full), s_i cannot carry out t's commit, and has to abort t.[2] This situation characterises a unilateral abort. The problem with unilateral aborts is that they are non-deterministic events, and thus, violate the assumption about the (deterministic) way requests are processed by database sites in the Database State Machine.

One way of coping with unilateral aborts is introducing a coordination phase (e.g., Atomic Commitment) before committing transactions. This solution introduces an additional cost in the transaction processing (additional communication between sites) which will only be justified in (hopefully rare) abnormal situations. Performance problems aside, introducing an Atomic Commitment phase in the state machine approach might have major implications. For example, the deterministic requirement on the manner requests are processed could be reconsidered.

Another way of dealing with unilateral aborts is treating them as site failures. In this case, as soon as the site recovers (e.g., in case of disk full, the recover procedure consists in allocating more disk space), the transaction is committed on that site.

[2] Note that in the Database State Machine, "local reasons" are not related to concurrency control.

This means that database site s_i will not be able to certify and commit any transaction t', $deliver(t) \prec deliver(t')$, until s_i is able to commit t (i.e., after the problem that prevents t from committing has been removed). It does not make much sense either to execute transactions locally at s_i before committing t, and, from the client's point of view, this behaviour is similar to a database site failure.

3.3 The Reordering Certification Test

Transactions running without any synchronisation between database sites may lead to high abort rates. In this section, we show how the certification test can be modified such that more transactions pass the certification test, and thus, do not abort.

3.3.1 Reordering Principle

The reordering certification test is based on the observation that the serial order in which transactions are committed does not need to be the same total order in which transactions are delivered to the certifier [PGS97]. The idea is to dynamically build a serial order (that does not necessarily follow the delivery order) in such a way that less aborts are produced. By being able to reorder a transaction t_a to a position other than the one t_a is delivered, the reordering protocol increases the probability of committing t_a.

The Database State Machine augmented with the Reordering technique differs from the Database State Machine presented in Section 3.2 in the way the certification test is performed for committing transactions (see Figure 3.5). The certifier distinguishes between committed transactions already applied to the database and committed transactions in the *Reorder List*. The Reorder List contains committed transactions whose write locks have been granted but whose updates have not been applied to the database yet, and thus, have not been seen by transactions in execution. The bottom line is that transactions in the Reorder List may change their relative order.

The number of transactions in the Reorder List is limited by a predetermined threshold, the *Reorder Factor*. Whenever the Reorder Factor is reached, the leftmost transaction t_a in the Reorder List is removed, its updates are applied to the database, and its write locks are released. If no transaction in the Reorder List is waiting to acquire a write lock just released by t_a, the corresponding data item is available to executing transactions. The reordering technique reduces the number of aborts, however, introduces some data contention since data items remain blocked longer. This expected tradeoff was indeed observed by our simulation model (see Section 3.5.3).

3.3.2 The Termination Protocol based on Reordering

Let $database_{s_i} = t_{(0)} \circ t_{(1)} \circ \cdots \circ t_{(last_{s_i}(\tau))}$ be the sequence containing all transactions on database site s_i at time τ that have passed the certification test augmented with the reordering technique (order of delivery plus some possible reordering). The

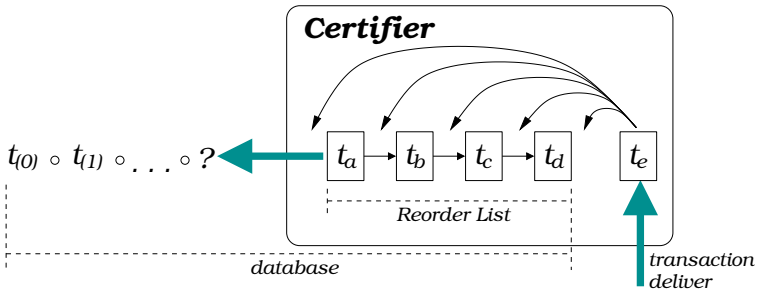

Figure 3.5: Reorder technique (reorder factor = 4)

sequence $database_{s_i}$ includes transactions that have been applied to the database and transactions in the Reorder List. We define $pos(t)$ the position transaction t has in $database_{s_i}$, and extend the termination protocol described in Section 3.2.3 to include the reordering technique.

1. *Certification test.* Database site s_j commits t_a if there is a position in the Reorder List where t_a can be inserted. Transaction t_a can be inserted in position p in the Reorder List if both following conditions are true.

   (a) For every transaction t_b in the Reorder List such that $pos(t_b) < p$, either t_b precedes t_a, or t_b has not updated any data item that t_a has read (this is essentially the certification test described in Section 3.1.3).

   (b) For every transaction t_b in the Reorder List such that $pos(t_b) \geq p$, (b.1) t_b does not precede t_a, or t_a has not read any data item written by t_b, and (b.2) t_a did not update any data item read by t_b.

   The certification test with reordering is formalised next as a state transition from the committing state to the committed state:

$$Committing(t_a, s_j) \rightsquigarrow Committed(t_a, s_j) \equiv$$

$$\left[\begin{array}{l} \exists \text{position } p \text{ in the Reorder List s.t. } \forall t_b,\ Committed(t_b, s_j) : \\ \\ pos(t_b) < p \Rightarrow t_b \rightarrow t_a \vee WS(t_b) \cap RS(t_a) = \emptyset \wedge \\ \\ pos(t_b) \geq p \Rightarrow \left(\begin{array}{c} (t_b \not\rightarrow t_a \vee WS(t_b) \cap RS(t_a) = \emptyset) \\ \wedge \\ WS(t_a) \cap RS(t_b) = \emptyset \end{array}\right) \end{array}\right]$$

   The condition for a transition from the committing state to the aborted state is the complement of the right side of this expression.

2. *Commitment.* If t_a passes the certification test, t_a is included in the Reorder List at position p, that is, all transactions in the Reorder List that are on the

right of p, including the one at p, are shifted one position to the right, and t_a is included. If, with the inclusion of t_a, the Reorder List reaches the Reorder Factor threshold, the leftmost transaction in Reorder List is removed and its updates are applied to the database.

3.3.3 Algorithm Correctness

From Lemma 3.1, every database site builds the same multiversion serialisation graph. It remains to show that all the histories produced by every database site using reordering have a multiversion serialisation graph that is acyclic, and, therefore, 1SR.

We redefine the version-order relation $\ll$ for the termination protocol based on reordering as follows: $x_a \ll x_b$ iff $pos(t_a) < pos(t_b)$ and $t_a, t_b \in MVSG_{s_i}$.

Lemma 3.4 *If there is a* read-from *relation* $t_a \hookrightarrow t_b \in MVSG_{s_i}$ *then* $pos(t_a) < pos(t_b)$.

PROOF: For a contradiction, assume that $t_a \hookrightarrow t_b \in MVSG_{s_i}$ and $pos(t_b) < pos(t_a)$. A read-from relation $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$ if $r_b[x_a] \in C(H)_{s_i}, a \neq b$, resulting in two cases of interest: (a) t_b was delivered and committed before t_a, and (b) t_b was delivered and committed after t_a but reordered to a position before t_a. The case in which t_a is delivered and committed after t_b is the same as case (a), and the case in which t_a is delivered before t_b and reordered to a position before t_b is not possible since when t_a is certified, t_b is not in the Reorder List.

In case (a), it follows that t_b reads uncommitted data (x_a) from t_a, which is not possible: if t_a and t_b executed at the same database site, reading uncommitted data is avoided by the strict 2PL rule, and if t_a and t_b executed at different database sites, t_a's updates are only seen by t_b after t_a's commit. In case (b), from the certification test augmented with reordering, when t_b is certified, we have that $(t_a \not\rightarrow t_b \vee WS(t_a) \cap RS(t_b) = \emptyset) \wedge WS(t_b) \cap RS(t_a) = \emptyset$ evaluates true. (Note that since t_b is the committing transaction, the indexes a and b in the expression given in the previous section have been inverted.) Since t_b reads-from t_a, $WS(t_a) \cap RS(t_b) \neq \emptyset$, and so, it must be that $t_a \not\rightarrow t_b$. If t_a and t_b executed at the same database site, $t_a \not\rightarrow t_b$ implies $committing(t_b) \xrightarrow{e} committing(t_a)$. However, this is only possible if t_b reads x from t_a before t_a commits, contradicting the strict 2PL rule. If t_a and t_b executed at different database sites, $t_a \not\rightarrow t_b$ implies $commit(t_a)_{site(t_b)} \overset{e}{\not\rightarrow} committing(t_b)$, and so, t_b passed to the committing state before t_a committed at $site(t_b)$, which contradicts the fact that t_b reads from t_a, and concludes the proof of the Lemma. □

Lemma 3.5 *If there is a* version-order *relation* $t_a \hookrightarrow t_b \in MVSG_s$ *then* $pos(t_a) < pos(t_b)$.

PROOF: According to the definition of version-order edges, there are two cases of interest. (1) Let $r_c[x_b]$, $w_b[x_b]$, and $w_a[x_a]$ be in $C(H)_{s_i}$ (a, b and c distinct), and

$x_a \ll x_b$, which implies $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$. It follows from the definition of version-order that $pos(t_a) < pos(t_b)$. (2) Let $r_a[x_c]$, $w_c[x_c]$, and $w_b[x_b]$ be in $C(H)_{s_i}$ (a, b and c distinct), and $x_c \ll x_b$, which implies $t_a \hookrightarrow t_b$ is in $MVSG_{s_i}$. We show that $pos(t_a) < pos(t_b)$. Since t_a reads-from t_c, t_c commits before t_a is certified, and there are two situations to consider.

(a) t_c and t_b have been committed when t_a is certified. Assume for a contradiction that $pos(t_b) < pos(t_a)$. From the certification test, we have that either $t_b \rightarrow t_a$ or $WS(t_b) \cap RS(t_a) = \emptyset$. Since $x \in WS(t_b)$ and $x \in RS(t_a)$, $WS(t_b) \cap RS(t_a) \neq \emptyset$, and so, it must be that $t_b \rightarrow t_a$. However, t_a reads x from t_c and not from t_b, which can only happen if $x_b \ll x_c$, contradicting that $x_c \ll x_b$.

(b) t_c and t_a have been committed when t_b is certified. Assume for a contradiction that $pos(t_b) < pos(t_a)$. From the certification test, it must be that $(t_a \not\rightarrow t_b \vee WS(t_a) \cap RS(t_b) = \emptyset) \wedge WS(t_b) \cap RS(t_a) = \emptyset$ evaluates true, which leads to a contradiction since $x \in WS(t_b)$ and $x \in RS(t_a)$, and therefore, $WS(t_b) \cap RS(t_a) \neq \emptyset$. □

Theorem 3.2 *Every history H produced by the Database State Machine algorithm augmented with the reordering technique is $1SR$.*

Proof: By Lemmas 3.4 and 3.5, every database site s_i produces a serialisation graph $MVSG^k_{s_i}$ such that every edge $t_a \hookrightarrow t_b \in MVSG^k_{s_i}$ satisfies the relation $pos(t_a) < pos(t_b)$. Hence, every database site produces an acyclic multiversion serialisation graph $MVSG^x_s$. By Lemma 3.1, every database site s_i constructs the same $MVSG^k_{s_i}$. By the Multiversion Serialisation Graph theorem of [BHG87], every history produced by the Database State Machine algorithm augmented with Reordering is 1SR. □

3.4 Simple Probabilistic Analysis

In this section, we evaluate the Database State Machine approach using a simple analytical model, based on the deadlock analysis presented in [GR93]. Our analytical model characterises the abort rate of the Database State Machine without the Reordering technique.

We simplify the analysis by making some assumptions about the system. The database is composed of DB data items, all with the same probability of being accessed (i.e., no hotspots). All transactions follow the same pattern, that is, each transaction executes a fixed number nr of read accesses and a fixed number nw of write accesses. Only update transactions are considered.

First, we calculate the probability that a transaction passes the certification test. If there are only two concurrent transactions in the system, t_a and t_b, and t_a commits before t_b, the probability that a read operation performed by t_b does not conflict with any write performed by t_a is $(1 - nw/DB)$, and the probability that no read performed by t_b conflicts with no write performed by t_a (i.e., the likelihood that t_b

passes the certification test) is $\prod_{i=0}^{nr-1}(1-nw/(DB-i))$. Considering that $DB \gg nr$ (much bigger than), this is approximately $(1-nw/DB)^{nr}$.

For a set G of N concurrent transactions, and assuming a worst case analysis where all transactions in G have non-intersecting write sets, the probability that the i-th transaction passes the certification test after the commit of $(i-1)$ transactions, denoted $P_{i,N}$, is

$$P_{i,N} = \left(1 - \frac{(i-1)\ nw}{DB}\right)^{nr}. \tag{3.1}$$

If we consider that $(i-1)\ nw \ll DB$ (much smaller than), expression (3.1) can be simplified as follows

$$\begin{aligned} P_{i,N} &= 1 - \binom{nr}{1}\left(\frac{(i-1)\ nw}{DB}\right) + \ldots + \binom{nr}{nr}\left(\frac{(i-1)\ nw}{DB}\right)^{nr} \\ &\approx 1 - \frac{(i-1)\ nr\ nw}{DB}, \end{aligned} \tag{3.2}$$

since the high-order terms in (3.2) can be dropped [GR93].

In the average, the probability P_C that a transaction t in G passes the certification test is

$$P_C \approx \frac{1}{N}\sum_{i=1}^{N} P_{i,N} = 1 - \frac{(N-1)\ nr\ nw}{2\ DB}. \tag{3.3}$$

Furthermore, considering TPS_{up} the number of update transactions submitted per second in the system, and τ the time in seconds it takes for a transaction to be delivered and certified, $N = TPS_{up}\ \tau$. However, not all transactions in G may cause t's abort since transactions that executed at the same site as t are properly ordered with t by local locking mechanisms (we assume that the probability of local deadlock is very small). Excluding such transactions leads to $N^* = TPS_{up}\ \tau\ (n_S - 1)/n_S$ (recall that n_S is the number of database sites). From (3.3), in the average, the likelihood that a transaction t does not pass the certification test, $\overline{P_C}$, is

$$\overline{P_C} \approx \frac{(N^*-1)\ nr\ nw}{2\ DB}. \tag{3.4}$$

Using $\overline{P_C}$ we can estimate the abort rate of the Database State Machine. In Section 3.5, we compare this probabilistic abort rated with results obtained with our simulation model.

3.5 Simulation Model

The simulation model we have developed abstracts the main components of a replicated database system (our approach is similar to [ACL87]). In this section, we

describe the simulation model, analyse the behaviour of the Database State Machine approach using the output provided by the simulation model, and compare some of the results obtained with the simulation model with the results obtained with the probabilistic analysis developed in the previous section.

3.5.1 Database Model and Settings

Every database site is modelled as a processor with some data disks and a log disk as local resources. The network is modelled as a common resource shared by all database sites. Each processor is shared by a set of execution threads, a terminating thread, and a workload generator thread (see Figure 3.6). All threads have the same priority, and resources are allocated to threads in a first-in-first-out basis. Each execution thread executes one transaction at a time, and the terminating thread is responsible for doing the certification. The workload generator thread creates transactions at the database site. Execution and terminating threads at a database site share the database data structures (e.g., lock table).

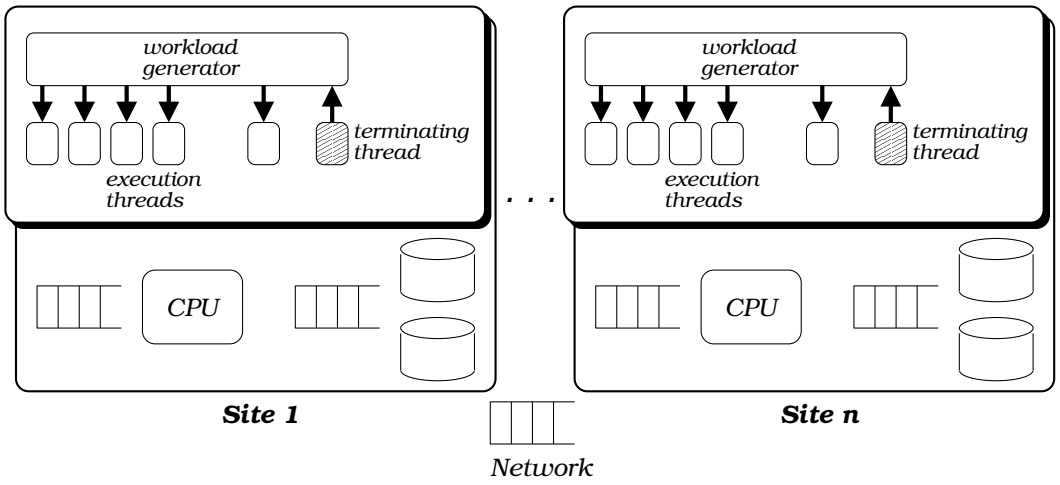

Figure 3.6: Simulation model

Committing transactions are delivered by the terminating thread and then certified. If a transaction passes the certification test, its write locks are requested and its updates are performed. However, once the terminating thread acquires the transaction's write locks, it makes a log entry for this transaction (with its writes) and assigns an execution thread to execute the transaction's updates over the database. This releases the terminating thread to treat the next committing transaction.

The parameters considered by our simulation model with the settings used in the experiments are shown in Table 3.1. The workload generator thread creates transactions and assigns them to executing threads according to the profile described (percentage of *update transactions*, percentage of *writes in update transactions*, and number of operations). We have chosen a relative small *database size* in order to reach data contention quickly and avoid extremely long simulation runs that would

be necessary to obtain statistically significant results.

We use a closed model, that is, each terminated transaction (committed or aborted) is replaced by a new one. Aborted transactions are sent back to the workload generator thread, and some time later resubmitted at the same database process where they originated. The *multiprogramming level* determines the number of executing threads at each database process. Local deadlocks are detected with a timeout mechanism: transactions are given a certain amount of time to execute (*transaction timeout*), and transactions that do not reach the committing state within the timeout are aborted.

Database parameters		**Processor parameters**	
Database size (data items)	2000	Processor speed	100 MIPS
Database sites (n)	1..8	Execute an operation	2800 instr.
Multiprogramming level (MPL)	8	Certify a transaction	5000 instr.
Data item size	2 KB	Reorder a transaction	15000 instr.
Transaction parameters		**Disk parameters** (Seagate ST-32155W)	
Update transactions	10%	Number of data disks	4
Writes in update transactions	30%	Number of log disks	1
Number of operations	5..15	Miss ratio	20%
Transaction timeout	0.5 sec	Latency	5.54 msec
Reorder factor	$0, n, 2n, 3n, 4n$	Transfer rate (Ultra-SCSI)	40 MB/sec
General parameters		**Communication parameters**	
Control data size	1 KB	Atomic Broadcasts per second	$\infty, 180, 800/n$
		Communication overhead	12000 instr.

Table 3.1: Simulation model parameters

Processor activities are specified as a number of instructions to be performed. The settings are an approximation from the number of instructions used by the simulator to execute the operations. The certification test is efficiently implemented by associating to each database item a version number [ACL87]. Each time a data item is updated by a committing transaction, its version number is incremented. When a transaction first reads a data item, it stores the data item's version number (this is the transaction read set). The certification test for a transaction consists thus in comparing each entry in the transaction's read set with the current version of the corresponding data item. If all data items read by the transaction are still current, the transaction passes the certification test. We consider that version numbers are stored in main memory. The reordering test is more complex, since it requires handling read sets and write sets of transactions in the reorder list. The *control data size* contains the data structures necessary to perform the certification test (e.g., readset and writeset). Atomic Broadcast settings are described in the next section.

3.5.2 Atomic Broadcast Implementation

We do not consider any specific Atomic Broadcast algorithm in our simulation. Instead, we take a more general approach, based on broadcast algorithms *classes*, according to scalability issues. Our simulation is based on these classes of algorithms.

Atomic Broadcast algorithms can be divided into two classes. We say that an Atomic Broadcast algorithm scales well, and belongs to the first class, if the number of messages delivered per time unit in the system is independent of the number of sites that deliver the messages. This class is denoted *class* k, where k determines the number of messages that can be delivered per time unit. An Atomic Broadcast algorithm of class k is presented in [Jal98]. In order to keep a constant delivery time, the algorithm in [Jal98] relies on special hardware.

If the number of messages delivered per time unit in the system decreases with the number of database sites that deliver the messages, the Atomic Broadcast algorithm does not scale well, and belongs to the second class. This class is denoted *class* k/n, where n is the number of sites that deliver the messages, and k/n is the number of messages that can be delivered per time unit. In this case, the more sites are added, the longer it takes to deliver a message, and so, the number of messages delivered in the system per time unit decreases exponentially with the number of sites. Most Atomic Broadcast algorithms fall in this category (e.g., [BSS91, CM84, CT96, GMS91, LG90, WS95]).

As a reference, we also define an Atomic Broadcast that delivers messages instantaneously. Such an algorithm is denoted *class* ∞ (i.e., it would allow in theory an infinite number of messages to be delivered per time unit).

The value chosen for class k/n in Table 3.1 is an approximation based on experiments with SPARC 20 workstations running Solaris 2.3 and an FDDI network (100Mb/s) using the UDP transport protocol with a message buffer of 20 Kbytes. The Atomic Broadcast algorithm used in the experiments is of class k/n, and the results found allowed to estimate $k = 800$ in k/n. The value for class k was arbitrarily taken as 180. Moreover, for all classes, the execution of an Atomic Broadcast has some *communication overhead* that does not depend on the number of sites (see Table 3.1).

3.5.3 Experiments and Results

In the following, we discuss the experiments we conducted and the results obtained with the simulation model. Each point plotted in the graphics has a confidence interval of 95%, and was determined from a sequence of simulations, each one containing 100000 submitted transactions. In order to remove initial transients [Jai91], only after the first 1000 transactions had been submitted, the statistics started to be gathered.

In some of the graphics presented next, we analyse update and read-only transactions separately, although the values presented were observed in the same simulations (i.e., all simulations contain update and read-only transactions).

Update Transactions Throughput. The experiments shown in Figures 3.7 and 3.8 evaluate the effects of the Atomic Broadcast algorithm classes on the transaction throughput. In these experiments, each cluster of database sites processed as many transactions as possible, that is, transaction throughput was only limited by the resources available. Figure 3.7 shows the number of update transactions submitted, and Figure 3.8 the number of update transactions committed. From Figure 3.7, the choice of a particular Atomic Broadcast algorithm class is not relevant for clusters with less than five database sites: whatever the class, transaction throughput increases linearly with the number of database sites. This happens because until four database sites, all three configurations are limited by the same resource, namely, local data disks (see paragraph about Resource Utilisation). Since the number of data disks increases linearly with the number of database sites, transaction throughput also increases linearly. For clusters with more than four database sites, contention is determined differently for each algorithm class. For class ∞, data contention prevents linear throughput growth, that is, for more than five sites, the termination thread reaches its limit and it takes much longer for update transactions to commit. The result is that data items remain locked for longer periods of time, impeding the progress of executing transactions. For classes k and k/n, contention is caused by the network (the limit being 180 and $800/n$ messages delivered per second, respectively).

It was expected that after a certain system load, the terminating thread would become a bottleneck, and transaction certification critical. However, from Figure 3.8, this only happens for algorithms of class ∞ (about 170 update transactions per second), since for algorithms in the other classes, the network becomes a bottleneck before the terminating thread reaches its processing limit. Also from Figure 3.8, although the number of transactions submitted per second for clusters with more than four sites is constant for class k, the number of transaction aborts increase as the number of database sites augments. This is due to the fact that the more database sites, the more transactions are executed under an optimistic concurrency control and thus, the higher the probability that a transaction aborts. The same phenomenon explains the difference between submitted and committed transactions for class k/n. For class ∞, the number of transactions committed is a constant, determined by the capacity of the terminating thread.

Queries Throughput. Figures 3.9 and 3.10 show submitted and committed queries per second in the system. The curves in Figure 3.9 have the same shape as the ones in Figure 3.7 because the simulator enforces a constant relation between submitted queries and submitted update transactions (see Figure 3.1, *update transactions* parameter). Update transactions throughput is determined by data and resource contention, and thus, queries are bound to exhibit the same behaviour. If update transactions and queries were assigned a fixed number of executing threads at the beginning of the simulation, this behaviour would not have been observed, however, the relation between submitted queries and update transactions would be determined by internal characteristics of the system and not by an input parameter, which would complicate the analysis of the data produced in the simulation. Queries are only aborted during their execution to solve local deadlocks they are involved in, or on behalf of committing update transactions that have passed the certification test and

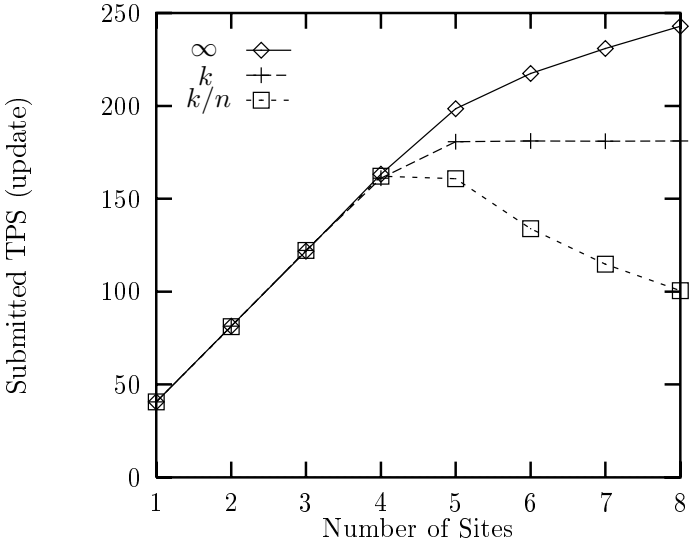

Figure 3.7: Submitted TPS (update)

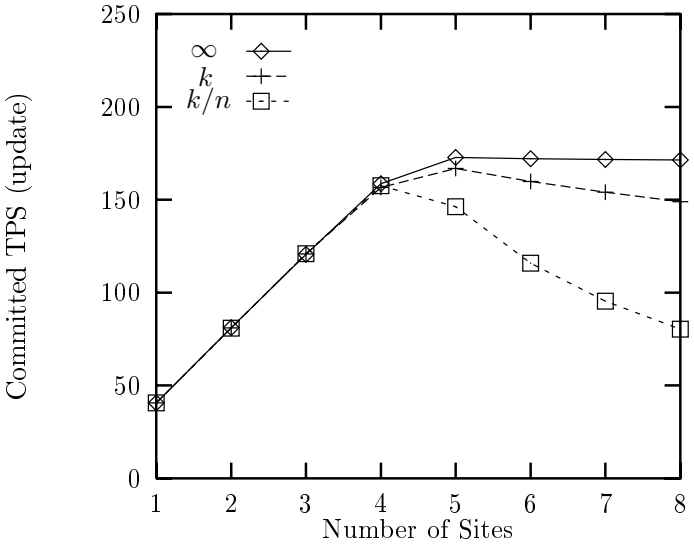

Figure 3.8: Committed TPS (update)

are requesting their write locks (Section 3.2.3). As shown in Figure 3.9 and 3.10, the values for submitted and committed queries, for all Atomic Broadcast algorithm classes, are very close to each other, which amounts to a small abort rate.

Reordering. Figures 3.11 and 3.12 show the abort rate for algorithms in the classes k and k/n respectively, with different reorder factors. We do not consider algorithms in the class ∞ because reordering does not bring any improvement to the abort rate in this case (even if more transactions passed the certification test, the terminating thread would not be able to process them). In both cases, reorder factors smaller than $4n$, have proved to reduce the number of aborted update transactions. For reordering factors equal to or greater than $4n$, the data contention introduced by the reordering technique leads to an increase on the abort rate that is greater than the reduction obtained with its use (i.e., the reordering technique increases the abort rate of update transactions). When the system reaches this point, most executing update and read-only transactions time out and are aborted by the system.

Abort Rate. Figures 3.13 and 3.14 present the detailed abort rate for the Database State Machine based on algorithms of class k/n without and with the Reordering technique (reorder factor equal to $3n$). Figures 3.13 and 3.14 are not in the same scale because the results shown differ from more than one order of magnitude. Figure 3.13 also shows the values obtained with the probabilistic model developed in Section 3.4. The graphics only include the aborts during transaction execution, and, in the case of update transactions, due to failing the certification test. Aborts due to time out are not shown because in the cases presented they amount to a small fraction of the abort rate. Without reordering (Figure 3.13), most transactions fail the certification test and are aborted.

The results observed are very close to those calculated using the probabilistic model. In order to draw the probabilistic curve, we have to calculate N, the number of concurrent transactions. N is expressed as the product of TPS_{up}, the number of update transactions submitted per second, and τ, the time it takes to execute a transaction. We take the values of TPS_{up} and τ from the simulation experiments.

When the Reordering technique is used, the number of transactions that fail the certification test is smaller than the number of transactions aborted during their execution (see Figure 3.14).

Response Time. Figure 3.15 presents the response time for the executions shown in Figures 3.7 and 3.8. The price paid for the higher throughput presented by algorithms of class ∞, when compared to algorithms of class k, is a higher response time. For algorithms in the class k/n, this only holds for less than 7 sites. When the number of transactions submitted per second is the same for all classes of Atomic Broadcast algorithms (see Figure 3.16), algorithms in class ∞ are faster. Queries have the same response time, independently of the Atomic Broadcast class. Note that configurations with less than three sites are not able to process 1000 transactions per second. This explains why update transactions executed in a single database site

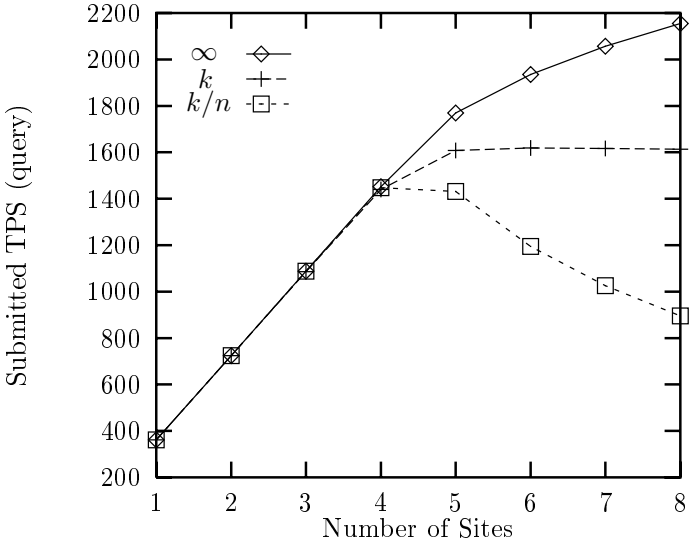

Figure 3.9: Submitted TPS (query)

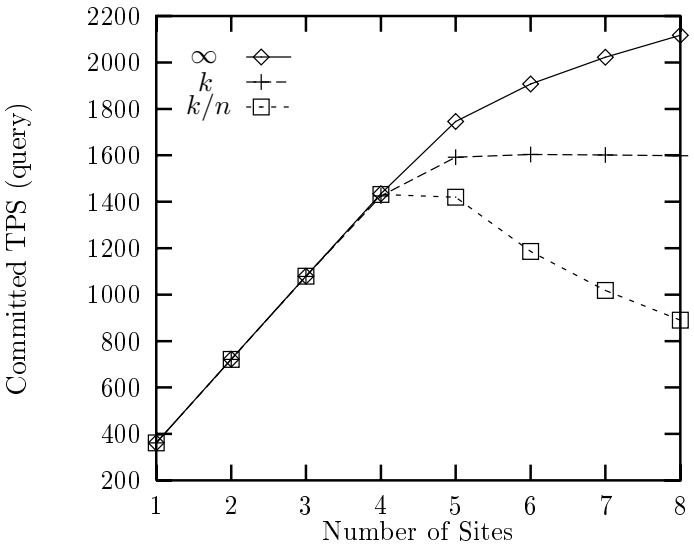

Figure 3.10: Committed TPS (query)

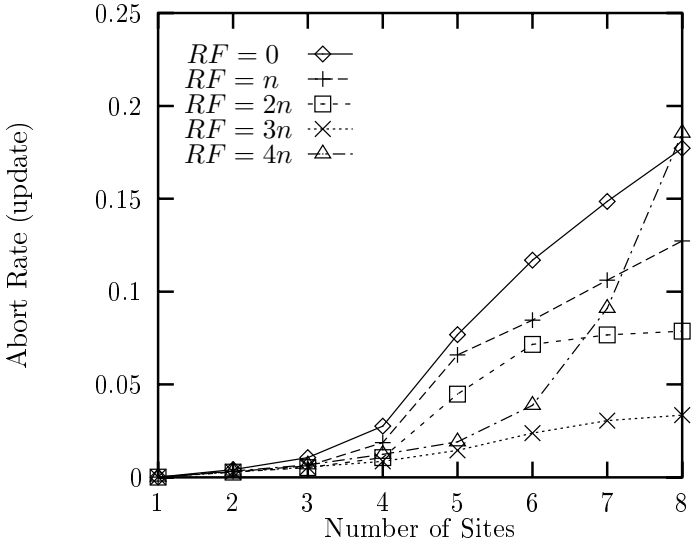

Figure 3.11: Reordering (class k)

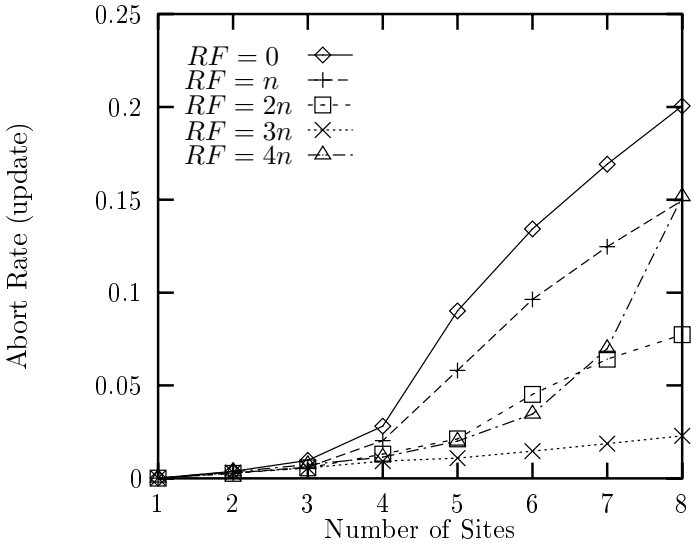

Figure 3.12: Reordering (class k/n)

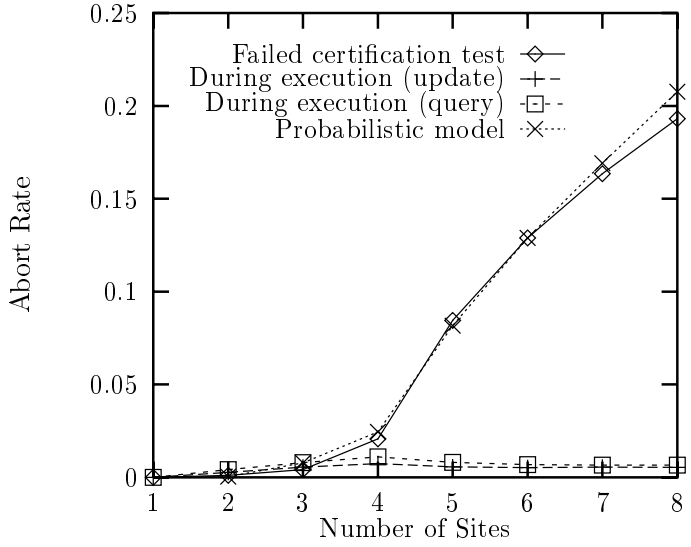

Figure 3.13: Abort rate (class k/n, $RF = 0$)

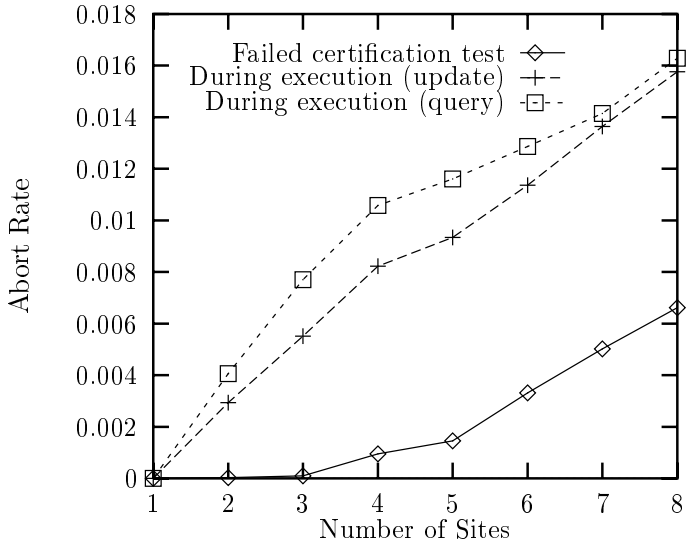

Figure 3.14: Abort rate (class k/n, $RF = 3n$)

have a better response time than update transactions executed in a Database State Machine with two sites (a single site reaches no more than 403 TPS, and a Database State Machine with two sites reaches around 806 TPS).

Figures 3.17 and 3.18 depict the degradation of the response time due to the Reordering technique. The increase in response time becomes accentuated when data contention becomes a problem (i.e., $RF = 4n$).

Resource Utilisation. Finally, Figures 3.19 and 3.20 present the way resource utilisation varies when the number of sites increases, with and without the Reordering technique. The values in Figure 3.19 were observed in the same experiments shown in Figures 3.7 – 3.10, and Figure 3.13, and the values in Figure 3.20 were observed in the same experiments depicted in Figures 3.14 and 3.18. In both cases, in a Database State Machine with less than five sites, the limiting resources are data disks. For five sites or more, the network becomes the bottleneck. The log disk utilisation curve has a shape similar to the curve that represents committed transactions, since the log is only used for committing transactions. This explains the superior log utilisation when the Reordering technique is used.

Overall Discussion. Besides showing the feasibility of the Database State Machine, the simulation model allows to draw some conclusions about its scalability. Update transactions scalability is determined by the scalability of the Atomic Broadcast algorithm class, which has showed to be a potential bottleneck of the system. This happens because the network is the only resource shared by all database sites (and network bandwidth does not increase as more database sites are added to the system). As for queries, only a slight grow in the abort rate was observed as the number of sites increase. This is due to the fact that queries are executed only locally, without any synchronisation among database sites.

The above result about update transactions scalability deserves a careful interpretation since, in regard to network resource utilisation, techniques that fully synchronise transactions between database sites (e.g., distributed 2PL protocol [BHG87]) probably will not outperform the Database State Machine. A typical Atomic Broadcast algorithm in the class k/n needs about $4n$ [PGS98] messages to deliver a transaction, and a protocol that fully synchronises transaction operations needs around $m \times n$ messages, where m is the number of transaction operations (assuming that reads and writes are synchronised) [BHG87]. Thus, unless transactions are very small ($m \leq 4$), the Database State Machine needs less messages than a technique that fully synchronises transactions.

Furthermore, the simulation model also shows that any effort to improve the scalability of update transactions should be concentrated on the Atomic Broadcast primitive. Finally, if on the one hand the deferred update technique has no distributed deadlocks, on the other hand its lack of synchronisation may lead to high abort rates. The simulation model has showed that, if well tuned, the reordering certification test can overcome this drawback.

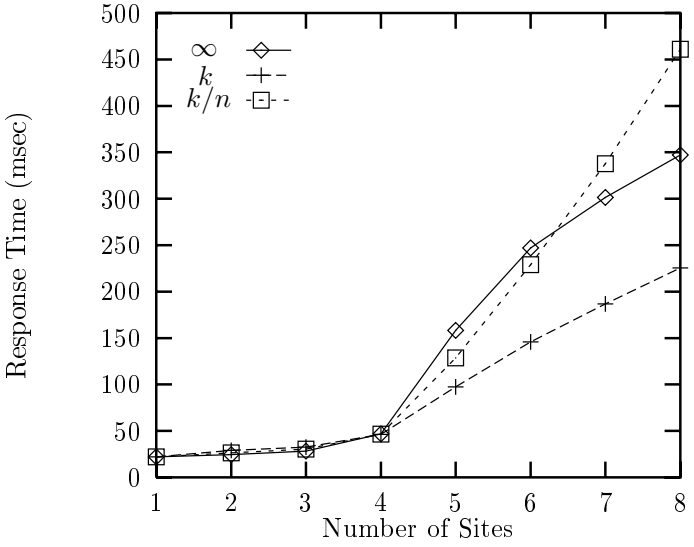

Figure 3.15: Response time *vs.* classes (update)

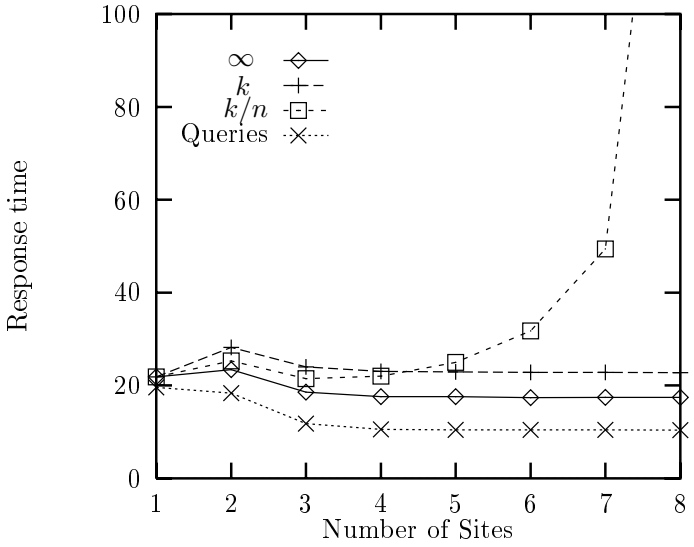

Figure 3.16: Response time ($TPS = 1000$)

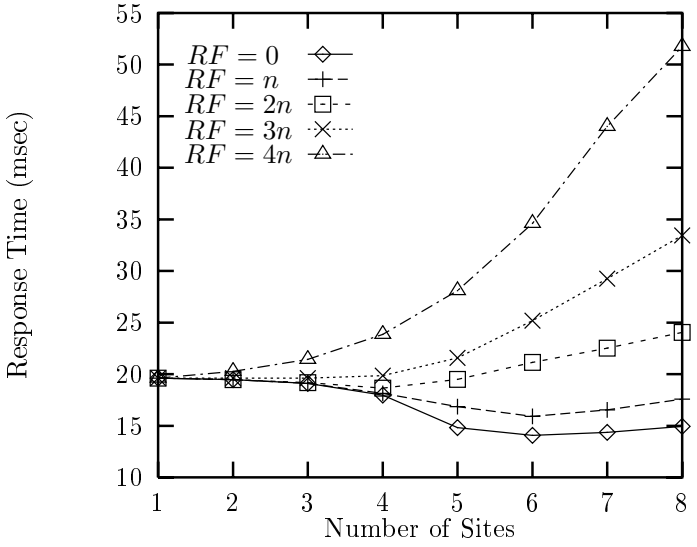

Figure 3.17: Response time *vs.* reordering (class k)

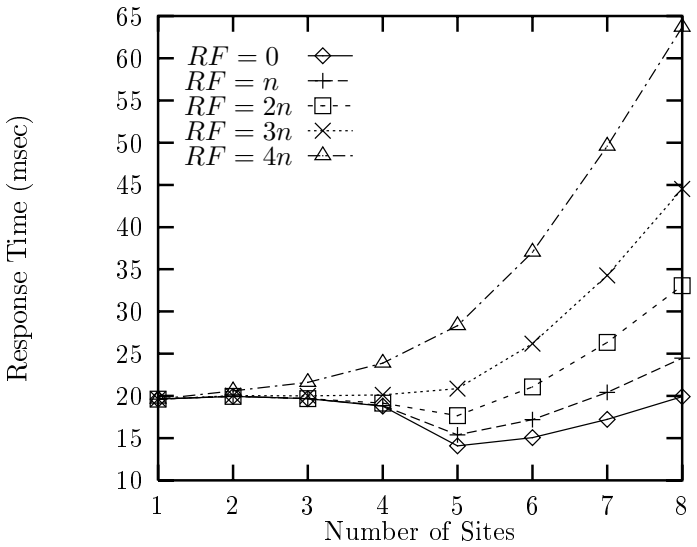

Figure 3.18: Response time *vs.* reordering (class k/n)

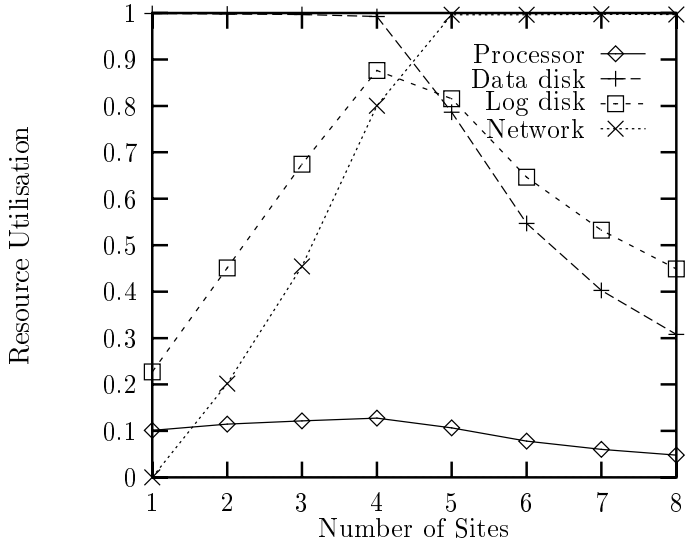

Figure 3.19: Resource Utilisation (class k/n, $RF = 0$)

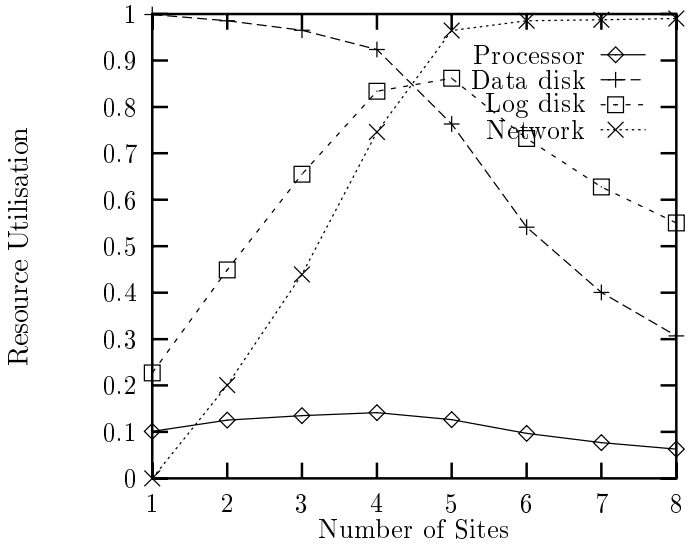

Figure 3.20: Resource Utilisation (class k/n, $RF = 3n$)

3.6 Related Work

The Database State Machine is an example of the deferred update technique. In this section, we situate the deferred update technique in the context of replicated databases and present replicated database algorithms that are related to the Database State Machine.

3.6.1 Database Replication

Database replication techniques can be classified according to the way updates are propagated to database sites, and the way updates are regulated. These two criteria define two orthogonal attributes that characterise database replication techniques.

Updates can be propagated in an eager or lazy way [GHOS96]. In *eager replication*, client update requests are applied to all correct database sites as part of the original transaction (i.e., the transaction commits in all correct database sites or in none). In *lazy replication*, a transaction first commits at one database site, and then the other database sites are updated as different transactions. Lazy replication may not preserve one-copy serialisability.

Eager replication admits two variations [BHG87]. *Immediate update replication* propagates every single client request to all database sites during the execution of the transaction, whereas in the *deferred update replication*, a single database site receives and processes all client requests, and only when the client requests the commit operation, the updates are propagated to the other database sites.

Master and group based replication regulates the way database sites accept update requests [GHOS96]. In the *master based replication*, only one database site can process update requests, while in the *group based replication*, any database site can receive update requests from the clients and modify the database. These mechanism can be considered as a kind of database ownership, where only the database owner has the right to process updates. In the master based replication there is only one database owner and in the group based replication there are n database owners. With Master based techniques, the failure of the Master prevents any update operation from being processes until the Master recovers. If availability is an important issue, then some mechanism is necessary to assign a new Master.

Table 3.2 summarises the attributes that characterise database replication protocols. The Database State Machine is an eager group replication mechanism.

Commercial databases have mostly focused on lazy replication techniques. This is in part due to the fact that commercial replication has sometimes other goals than high availability (e.g., replication may aim only at performance, or providing support for off-line analytical processing).

Sybase Replication Server [Syb99] and IBM Data Propagator [Gol95], are examples of master based lazy replication. Although these mechanisms are implemented in different ways,[3] they both share the particularity that replication is implemented

[3]Sybase Replication Server is based on the "push model," where sites subscribe to copies of data,

	Eager Replication	**Lazy Replication**
Group Ownership	N transactions N Database owners	1 transaction N Database owners
Master Ownership	N transactions 1 Database owner	1 transaction 1 Database owner

Table 3.2: Database replication classification

"outside the database engine," and in both cases, the replication mechanism interferes as little as possible in the "normal" (i.e., without replication) execution. Oracle version 7.1 offers mechanisms to implement any replication strategy [Del95]. To keep database consistency with lazy group and lazy master replication, Oracle provides conciliation rules that can be used to solve conflicts [Ora95, Jac95].

In the next sections, we present other database replication proposals. This is a difficult task to accomplish due to the multitude of replicated database algorithms and the variety of assumptions that they make about the system. Thus, before proceeding with our discussion, we point out that the Database State Machine is at the intersection of two axes of research. First, relying on a certification test to commit transactions is an application of optimistic concurrency control. However, terminating transactions with an Atomic Broadcast primitive is an alternative to solutions based on Atomic Commitment protocols. Furthermore, we mainly concentrate our discussion on eager group based replication.

3.6.2 Optimistic Concurrency Control

Although most commercial database systems are based on (pessimistic) 2PL synchronisation [GR93], optimistic concurrency control have received increasing attention since it introduction in [KR81] (see [Tho98, Bha99, OV99] for surveys). It has been shown in [ACL87] that if sufficient hardware resources are available, optimistic concurrency control can offer better transaction throughput than 2PL. This result is explained by the fact that an increase in the multiprogramming level, in order to reach high transaction throughput, also increases locking contention, and thus, the probability of transaction waits due to conflicts, and transaction restarts to solve deadlocks. The study in [ACL87] is for a centralised single-copy database. One could expect that in a replicated database, the cost of synchronising distributed accesses by message passing would be non negligible as well. In fact, the study in [GHOS96] has shown that fully synchronising accesses in replicated database contexts (as required by 2PL) is *dangerous*, since the probability of deadlocks is directly proportional to the third power of the number of database sites in the system.

and changes are propagated from the primary to the backups as soon as they occur, and IBM Data Propagator is based on the "pull model," where replicated data is demanded by the backups to the primary at regular time intervals.

3.6.3 Transaction Termination

The limitations of traditional Atomic Commitment protocols in replicated contexts have been recognised by many authors, and several algorithms have been proposed to terminate transactions in replicated databases without Atomic Commitment. However, most approaches are not eager group based, or require explicit use of application semantics.

The fact that Atomic Commitment leads to abort transactions in situations where a single replica manager crashes has been pointed out in [GOS96]. The authors propose a variation of the three phase commit protocol [Ske81] that commits transactions as long as a majority of replica managers are up.

In [DGH+87], lazy based *epidemic replication protocols* are proposed as an alternative to traditional replication protocols. Another example of epidemic replication is proposed in [JMR97], which relies on semantic knowledge about the application. Bayou [TTP+95] implements a lazy master replication mechanism, offering weak consistency, while the work in [BK97] presents a lazy master approach that guarantees one-copy serialisability.

A deferred update replication protocol that guarantees one-copy serialisability is presented in [AAS97]. In this protocol, transactions that execute at the same process share the same data items, using locks to solve local conflicts. This protocol is based on a variation of the three phase commit protocol to certificate and terminate transactions.

It is only recently that Atomic Broadcast has been considered as a possible candidate for terminating transactions in replicated databases. Schiper and Raynal [SR96] pointed out some similarities between the properties of Atomic Broadcast and static transactions (e.g., transactions whose operations are known in advance). Atomically broadcasting transactions was also addressed in [Kei94], which assumes that transaction operations are known at the beginning of the transaction execution. The work in [BK98] investigates relaxed isolation guarantees in order to develop database replication protocols.

In [AAAS97], a family of protocols for the management of replicated database based on the immediate and the deferred techniques is proposed. The immediate update replication consists in atomically broadcasting every write operation to all database sites. This mechanism requires that every database site execute each transaction operation in the same way. For the deferred update replication, two Atomic Broadcasts are necessary to commit a transaction. An alternative solution is also proposed, using a sort of multiversion mechanism to deal with the writes during transaction execution (if a transaction writes a data item, a later read should reflect this write).

Amir et al. [ADMSM94] also use Atomic Broadcast to implement replicated databases. However, the scheme proposed considers that clients submit individual object operations rather than transactions.

3.7 Discussion

The Database State Machine is an aggressive approach to building high performance replicated databases. Its principle is to reduce synchronisation between database sites to the utmost, requiring a deterministic transaction processing. Deterministic transaction processing is a delicate issue in the context of a replicated database. We have chosen to base the Database State Machine on the deferred update replication technique because this allowed us to concentrate the deterministic requirements on a very precise part of the system, the certification test. Furthermore, the deferred update replication technique also permits a fair distribution of load among database sites, that is, transactions are only executed at one database site, although update transactions are committed in all database sites.

The optimistic way in which transactions are processed in the deferred update replication may lead to high abort rates. The Database State Machine reduces the number of aborts using the Reordering technique, which exploits the serialisability property to commit transactions that otherwise would be aborted. Reordering transactions increases the response time, but, as it was observed in our simulations, for certain reorder factors, this cost is acceptable.

Some issues about the Database State Machine remain open for further studies. For example, we have not been concerned by the way clients choose the database site that will execute their requests. This is an important issue for load balancing. Another interesting issue for future studies is how to pass from full replication to partial replication. At first glance, this seems not to be possible because of the certification test. However, if the certification test is augmented with Atomic Commitment (see Section 3.2.5), partial replication becomes possible. Note that even if the certification test relies on an Atomic Commitment, propagating committing transactions with an Atomic Broadcast is still attractive since it increases the chance of committing transactions [PGS98].

Chapter 4

Generic Broadcast

All generalizations are dangerous,
even this one.

Alexandre Dumas

The Database State Machine relies on an Atomic Broadcast primitive to propagate update transactions. As shown in the previous chapter, Atomic Broadcast is sufficient to ensure the Database State Machine correctness, however, it turns out that it is not necessary. In this chapter, we introduce the Generic Broadcast problem, a broadcast primitive that allows applications to tailor-make their order requirements. The intuition behind Generic Broadcast is that message ordering has a cost, and for several applications, like the Database State Machine, total ordering of messages is stronger than necessary to guarantee correctness. Generic Broadcast allows the application to define a conflict relation that reflects the semantic meaning of the messages.

In addition to introducing the Generic Broadcast problem, this chapter presents an algorithm that solves it, and compares this algorithm to known Atomic Broadcast implementations using the delivery latency parameter. This chapter is based on an asynchronous system model. Processes communicate by message passing through Quasi-Reliable channels, and have the crash-stop mode of failure. The system is augmented with failure detectors (see Chapter 2).

4.1 Problem Definition

Generic Broadcast is defined by the primitives g-Broadcast and g-Deliver.[1] When a process p invokes g-Broadcast with a message m, we say that p g-Broadcasts m, and when p returns from the execution of g-Deliver with message m, we say that p g-Delivers m. Message m is taken from a set $\mathcal{M}$ to which all messages belong.

[1] g-Broadcast has no relation with the GBCAST primitive defined in the Isis system [BJ87].

Central to Generic Broadcast is the definition of a (symmetric) conflict relation on $\mathcal{M} \times \mathcal{M}$ denoted by $\mathcal{C}$ (i.e., $\mathcal{C} \subseteq \mathcal{M} \times \mathcal{M}$). If $(m, m') \in \mathcal{C}$ then we say that m and m' conflict. Generic Broadcast is specified by (1) a conflict relation $\mathcal{C}$ and (2) the following conditions:

> (VALIDITY) If a correct process g-Broadcasts a message m, then it eventually g-Delivers m.
>
> (AGREEMENT) If a correct process g-Delivers a message m, then all correct processes eventually g-Deliver m.
>
> (INTEGRITY) For any message m, every correct process g-Delivers m at most once, and only if m was previously g-Broadcast by some process.
>
> (PARTIAL ORDER) If correct processes p and q both g-Deliver messages m and m', and m and m' conflict, then p g-Delivers m before m' if and only if q g-Delivers m before m'.

The conflict relation $\mathcal{C}$ determines the pair of messages that are sensitive to order, that is, the pair of messages for which the g-Deliver order should be the same at all processes that g-Deliver the messages. The conflict relation $\mathcal{C}$ renders the above specification *generic*, as shown in the next section.

4.1.1 Instances of Generic Broadcast

We consider in the following some instances of Generic Broadcast. In particular, we show (a) that Reliable Broadcast and Atomic Broadcast are special cases of Generic Broadcast, (b) how Generic Broadcast can be defined in a scenario where operations commute, and (c) how Generic Broadcast can be defined in the context of the Database State Machine (see Chapter 3).

Reliable and Atomic Broadcast. Two special cases of conflict relations are the (1) empty conflict relation, denoted by $\mathcal{C}_\emptyset$, where $\mathcal{C}_\emptyset = \emptyset$, and the (2) $\mathcal{M} \times \mathcal{M}$ conflict relation, denoted by $\mathcal{C}_{\mathcal{M}\times\mathcal{M}}$, where $\mathcal{C}_{\mathcal{M}\times\mathcal{M}} = \mathcal{M} \times \mathcal{M}$. In case (1) no pair of messages conflict, that is, the partial order property imposes no constraint. This is equivalent to having only the validity, agreement, and integrity properties, which is called *Reliable Broadcast*. In case (2) any pair (m, m') of messages conflict, that is, the partial order property imposes that all pairs of messages be ordered, which is called *Atomic Broadcast*. In other words, Reliable Broadcast and Atomic Broadcast lie at the two ends of the spectrum defined by Generic Broadcast. In between, any other conflict relation defines an instance of Generic Broadcast.

Commuting Operations. Conflict relations lying in between the two extremes of the conflict spectrum can be better illustrated by an example. Consider a replicated *Account* object, defined by the operations *deposit(x)* and *withdraw(x)*. Clearly,

deposit operations commute with each other, while *withdraw* operations do not, neither with each other nor with *deposit* operations.[2] Let $\mathcal{M}_{deposit}$ denote the set of messages that carry a *deposit* operation, and $\mathcal{M}_{withdraw}$ the set of messages that carry a *withdraw* operation. This leads to the following conflict relation $\mathcal{C}_{Account}$:

$$\mathcal{C}_{Account} = \{ (m, m') \ : \ m \in \mathcal{M}_{withdraw} \textbf{ or } m' \in \mathcal{M}_{withdraw} \}.$$

Generic Broadcast with the $\mathcal{C}_{Account}$ conflict relation for broadcasting the invocation of deposit and withdraw operations to the replicated *Account* object defines a weaker ordering primitive than Atomic Broadcast (e.g., messages in $\mathcal{M}_{deposit}$ are not required to be ordered with each other), and a stronger ordering primitive than Reliable Broadcast (which imposes no order at all).

The Database State Machine Semantics. The termination protocol of the Database State Machine is based on an Atomic Broadcast primitive (Section 3.2.2). Once a transaction passes to the committing phase, its updates, read and write sets are atomically broadcast to all databases sites to be certified. Atomic Broadcast is sufficient to guarantee replica correctness, as it was shown in Section 3.2.4, however, it turns out that it is not necessary.

The following example shows that Atomic Broadcast is stronger than necessary to guarantee replica correctness. Assume two messages m and m' that transport two concurrent transactions t_a and t_b, denoted by $m : t_a$ and $m' : t_b$ respectively, such that $RS_a = \{x, y\}$ and $WS_a = \{z\}$, and $RS_b = \{x\}$ and $WS_b = \{w\}$. In this case, neither t_a (if committed) has any influence on the outcome of the certification test of t_b, nor t_b (if committed) has any influence on the outcome of the certification test of t_a. To see why, notice that $RS_a \cap WS_b = \emptyset$ and $RS_b \cap WS_a = \emptyset$. Furthermore, since both transactions have disjoint write sets, even if both are committed, the order their updates are performed in the database does not matter. Therefore, total order delivery of messages m and m' is not necessary for the Database State Machine to be correct.

A conflict relation $\mathcal{C}_{DSM}$, weaker than Atomic Broadcast, can be derived for the Database State Machine from the certification test, which checks whether transactions can be committed or not. The conflict relation $\mathcal{C}_{DSM}$ is defined as follows.

$$\mathcal{C}_{DSM} = \{ (m : t_a, m' : t_b) : (RS_a \cap WS_b \neq \emptyset) \vee (WS_a \cap RS_b \neq \emptyset) \vee (WS_a \cap WS_b \neq \emptyset) \}.$$

4.1.2 Strict Generic Broadcast Algorithm

From the specification it is obvious that any algorithm solving Atomic Broadcast also solves any instance of the Generic Broadcast problem defined by $\mathcal{C} \subseteq \mathcal{M} \times \mathcal{M}$. However, such a solution also orders messages that do not conflict. We are interested

[2] This is the case for instance if we consider that a *withdraw(x)* operation can only be performed if the current balance is larger than or equal to x.

in a *strict* algorithm, that is, an algorithm that does not order two messages if not required, according to the conflict relation $\mathcal{C}$. The idea is that ordering messages has a cost (e.g., in terms of number of communication steps) and this cost should be kept as low as possible. More formally, we define an algorithm that solves Generic Broadcast for a conflict relation $\mathcal{C} \subset \mathcal{M} \times \mathcal{M}$, denoted by $A_{\mathcal{C}}$, *strict* if it satisfies the condition below.

> (STRICTNESS) Consider an algorithm $A_{\mathcal{C}}$, and let $\mathcal{R}_{\mathcal{C}}^{NC}$ be the set of runs of $A_{\mathcal{C}}$. There exists a run R in $\mathcal{R}_{\mathcal{C}}^{NC}$, in which at least two correct processes g-Deliver two non-conflicting messages m and m' in a different order.

Informally, the strictness condition requires that algorithm $A_{\mathcal{C}}$ allow runs in which the g-Deliver of non conflicting messages is not totally ordered. However, even if $A_{\mathcal{C}}$ does not order messages, it can happen that total order is spontaneously ensured. So we cannot require violation of total order to be observed in every run: we require it in at least one run of $A_{\mathcal{C}}$.

4.2 Solving Generic Broadcast

In this section, we present an algorithm that solves Generic Broadcast. Our solution relies on an algorithm that solves the Consensus problem. We first provide an overview of the solution, and then present a detailed algorithm.

4.2.1 Overview of the Algorithm

Processes executing our Generic Broadcast algorithm progress in a sequence of stages numbered $1, 2, \ldots, k, \ldots$. Stage k terminates only if two conflicting messages are g-Broadcast, but not g-Delivered in some stage $k' < k$.

g-Delivery of non-conflicting messages. Let m be a g-Broadcast message. When some process p receives m in stage k, and m does not conflict with some other message m' already received by p in stage k, p inserts m in its $pending_p^k$ set, and sends an *ACK(m)* message to all processes. As soon as p receives *ACK(m)* messages from n_{ack} processes, where

$$n_{ack} \geq (n+1)/2, \tag{4.1}$$

p g-Delivers m.

g-Delivery of conflicting messages. Consensus is launched to terminate stage k if a conflict is detected. The Consensus decides on two sets of messages, denoted by $NCmsgSet^k$ (NC stands for Non-Conflicting) and $CmsgSet^k$ (C stands for Conflicting). The set $NCmsgSet^k \cup CmsgSet^k$ is the set of all messages g-Delivered in stage k. Messages in $NCmsgSet^k$ are g-Delivered before messages in $CmsgSet^k$,

and messages in $NCmsgSet^k$ may be g-Delivered by some process p in stage k before p executes the k-th Consensus. The set $NCmsgSet^k$ does not contain conflicting messages, while messages in $CmsgSet^k$ may conflict. Messages in $CmsgSet^k$ are g-Delivered in some deterministic order. Process p starts stage $k+1$ once it has g-Delivered all messages in $CmsgSet^k$.

Properties. To be correct, our algorithm must satisfy the following properties:

(a) If two messages m and m' conflict, then at most one of them is g-Delivered in stage k before Consensus.

(b) If message m is g-Delivered in stage k by some process p before Consensus, then m is in the set $NCmsgSet^k$.

(c) The set $NCmsgSet^k$ does not contain any conflicting messages.[3]

We discuss each of these properties informally. The formal proof of the algorithm is in Section 4.2.3. Property (a) is ensured by condition (4.1). Property (b) is ensured as follows. Before starting Consensus, every process p sends its $pending_p^k$ set to all processes (in a message of type *checking*, denoted by CHK), and waits for messages of type CHK from exactly n_{chk} processes. Only if some message m is at least in $\lceil (n_{chk}+1)/2 \rceil$ messages of type CHK, then m is inserted in $majMSet_p^k$, the initial value of Consensus that decides on $NCmsgSet^k$. So, if m is in less than $\lceil (n_{chk}+1)/2 \rceil$ messages of type CHK, m is not inserted in $majMSet_p^k$. Indeed, if condition

$$2n_{ack} + n_{chk} \geq 2n + 1 \tag{4.2}$$

holds and m is in less than $\lceil (n_{chk}+1)/2 \rceil$ messages of type CHK, then m could not have been g-Delivered in stage k before Consensus. To understand why, notice that from (4.2) and the fact that n_{ack}, n_{chk}, and $n \in \mathbb{N}$, we have (see Proposition 4.1)

$$(n - n_{chk}) + \lceil (n_{chk}+1)/2 \rceil \leq n_{ack}, \tag{4.3}$$

where $(n - n_{chk})$ is the number of processes from which p knows nothing. From (4.3), if m is in less than $\lceil (n_{chk}+1)/2 \rceil$ messages of type CHK, then even if all processes from which p knows nothing had sent $ACK(m)$, there would not be enough $ACK(m)$ messages to have m g-Delivered by some process in stage k before Consensus.

Proposition 4.1 *If* $2n_{ack} + n_{chk} \geq 2n + 1$ *and* n_{ack}, n_{chk}, *and* $n \in \mathbb{N}$ *then* $(n - n_{chk}) + \lceil (n_{chk}+1)/2 \rceil \leq n_{ack}$.

Proof: Solving $2n_{ack} + n_{chk} \geq 2n+1$ for n_{ack}, we have $n + (1 - n_{chk})/2 \leq n_{ack}$. But $n + (1 - n_{chk})/2 = (n - n_{chk}) + (n_{chk}+1)/2$, and so, $(n - n_{chk}) + (n_{chk}+1)/2 \leq n_{ack}$. We

[3] Property (c) does not follow from (a) and (b). Take for example two messages m and m' that conflict, but are g-Delivered in stage k as the result of the Consensus terminating stage k: neither property (a), nor property (b) applies.

claim that $(n-n_{chk})+(1+n_{chk})/2 \leq n_{ack}$ implies $(n-n_{chk})+\lceil(n_{chk}+1)/2\rceil \leq n_{ack}$. If n_{chk} is odd, the claim follows directly. Thus, assume that n_{chk} is even, that is, there is an $l \in \mathbb{N}$, such that $n_{chk} = 2\,l$. We have to show that $(n-2\,l)+(l+1/2) \leq n_{ack}$ implies $(n-2\,l)+\lceil(l+1/2)\rceil \leq n_{ack}$. Since $l \in \mathbb{N}$, $(n-2\,l)+(l+1/2) \leq n_{ack}$ implies $(n-2\,l)+(l+1) \leq n_{ack}$, and it is not difficult to see that if $(n-2\,l)+(l+1) \leq n_{ack}$ then $(n-2\,l)+\lceil(l+1/2)\rceil \leq n_{ack}$. □

Property (c) is ensured by the fact that m is inserted in $majMSet_p^k$ only if m is in at least $\lceil(n_{chk}+1)/2\rceil$ messages of type CHK received by p (majority condition). Let m and m' be two messages in $majMSet_p^k$. By the majority condition, the two messages are in the $pending_q^k$ set of at least one process q. This is however only possible if m and m' do not conflict.

Minimal number of correct processes. Our Generic Broadcast algorithm waits for n_{ack} messages before g-Delivering non-conflicting messages, and n_{chk} messages if a conflict is detected before starting Consensus. Therefore, our algorithm requires $\max(n_{ack}, n_{chk})$ correct processes. The minimum of correct processes to solve Generic Broadcast with our algorithm is $(2n+1)/3$, which happens when $n_{ack} = n_{chk}$.

4.2.2 Detailed Algorithm

Provided that the number of correct processes is at least $\max(n_{ack}, n_{chk})$, $n_{ack} \geq (n+1)/2$, and $2n_{ack}+n_{chk} \geq 2n+1$, Algorithm 1 solves Generic Broadcast for any conflict relation $\mathcal{C}$. All tasks in Algorithm 1 execute concurrently, and Task 3 has two entry points (lines 12 and 31).

Algorithm 1 uses an "underline" notation (e.g., $\underline{k}$) to precise the message a process is waiting for. For example, a process that waits for message $(\underline{k}, pending_q^k, \underline{ACK})$ (line 31) will receive a message $(i, -, type)$ such that $i = k$ and $type = ACK$.

Process p in stage k manages the following sets.

- $R\_delivered_p$: contains all messages R-delivered by p up to the current time,
- $G\_delivered_p$: contains all messages g-Delivered by p in all stages $k' < k$,
- $pending_p^k$: contains every message m such that p has sent an ACK message for m in stage k up to current time, and
- $localNCg\_Deliver_p^k$: is the set of non conflicting messages that are g-Delivered by p in stage k, up to the current time (and before p executes the k-th Consensus).

When p wants to g-Broadcast message m, p executes *R-broadcast*(m) (line 8). After R-delivering a message m, the actions taken by p depend on whether m conflicts or not with some other message m' in $R\_delivered_p \setminus G\_delivered_p$.

No conflict. If no conflict exists, then p includes m in $pending_p^k$ (line 14), and sends an ACK message to all processes, acknowledging the R-delivery of m (line 15). Once p receives n_{ack} ACK messages for a message m (line 31), p includes m in $localNCg\_Deliver_p^k$ (line 35) and g-Delivers m (line 36).

Conflict. In case of conflict, p starts the terminating procedure for stage k. Process p first sends a message of the type $(k, pending_p^k, CHK)$ to all processes (line 17), and waits the same information from exactly n_{chk} processes (line 18). Then p builds the set $majMSet_p^k$ (line 20).[4] It can be proved that $majMSet_p^k$ contains every message m such that for any process q, $m \in localNCg\_Deliver_q^k$. Then p starts Consensus (line 21) to decide on a pair $(NCmsgSet^k, CmsgSet^k)$ (line 22). Once the decision is made, process p first g-Delivers (in any order) the messages in $NCmsgSet^k$ that is has not g-Delivered yet (lines 23 and 25), and then p g-Delivers (in some deterministic order) the messages in $CmsgSet^k$ that it has not g-Delivered yet (lines 24 and 26). After g-Delivering all messages decided in Consensus execution k, p starts stage $k+1$ (lines 28-30).

4.2.3 Proof of Correctness

We first establish some Lemmata that will be used to prove the main result (i.e., Properties 4.2 – 4.5). Lemma 4.1 states that the set $pending^k$ does not contain conflicting messages. It is used to prove Lemmata 4.2 and 4.5.

Lemma 4.1 *For any process p, and all $k \geq 1$, if messages m and m' are in $pending_p^k$, then m and m' do not conflict.*

PROOF: Suppose, by way of contradiction, that there is a process p, and some $k \geq 1$ such that m and m' conflict and are in $pending_p^k$. Since m and m' are in $pending_p^k$, p must have R-delivered m and m'. Assume that p first R-delivers m and then m'. Thus, there is a time t after p R-delivers m' such that p evaluates the *if* statement at line 13, and $m' \in R\_delivered_p$, $m' \notin G\_delivered_p$, and $m' \notin pending_p^k$. At time t, $m \in R\_delivered_p$ (by the hypothesis m is R-delivered before m'), and $m \notin G\_delivered_p$ (if $m \in G\_delivered$, from lines 27-29 m and m' cannot be both in $pending_p^k$). Therefore, when the *if* statement at line 13 is evaluated, m and m' are in $R\_delivered \setminus G\_delivered$, and since m and m' conflict, the condition evaluates false, and m' is not included in $pending_p^k$, a contradiction that concludes the proof. □

Lemma 4.2 proves property (a) of page 57.

Lemma 4.2 *If two messages m and m' conflict, then at most one of them is g-Delivered in stage k before Consensus.*

[4] $majMSet_p^k = \{m : |Chk_p^k(m)| \geq (n_{chk} + 1)/2\}$

Algorithm 1 Generic Broadcast algorithm

1: Initialisation:
2: $R\_delivered \leftarrow \emptyset$
3: $G\_delivered \leftarrow \emptyset$
4: $k \leftarrow 1$
5: $pending^1 \leftarrow \emptyset$
6: $localNCg\_Deliver^1 \leftarrow \emptyset$

7: To execute *g-Broadcast*(m): *{**Task 1**}*

8: *R-broadcast*(m)

9: *g-Deliver*$(-)$ occurs as follows:

10: **when** *R-deliver*(m) *{**Task 2**}*
11: $R\_delivered \leftarrow R\_delivered \cup \{m\}$

12: **when** $(R\_delivered \setminus G\_delivered) \setminus pending^k \neq \emptyset$ *{**Task 3**}*
13: **if** [for all $m, m' \in R\_delivered \setminus G\_delivered,\ m \neq m' : (m, m') \notin Conflict$] **then**
14: $pending^k \leftarrow R\_delivered \setminus G\_delivered$
15: send$(k, pending^k, ACK)$ to all
16: **else**
17: send$(k, pending^k, CHK)$ to all
18: **wait until** [for n_{chk} processes $q : p$ received $(\underline{k}, pending_q^k, \underline{CHK})$ from q]
19: *#Define* $chkPSet^k(m) = \{q : p$ received $(\underline{k}, pending_q^k, \underline{CHK})$ from q **and** $m \in pending_q^k\}$
20: $majMSet^k \leftarrow \{m : |chkPSet^k(m)| \geq \lceil (n_{chk} + 1)/2 \rceil\}$
21: $propose(k, (majMSet^k, (R\_delivered \setminus G\_delivered) \setminus majMSet^k))$
22: **wait until** $decide(\underline{k}, (NCmsgSet^k, CmsgSet^k))$
23: $NCg\_Deliver^k \leftarrow (NCmsgSet^k \setminus localNCg\_Deliver^k) \setminus G\_delivered$
24: $Cg\_Deliver^k \leftarrow CmsgSet^k \setminus G\_delivered$
25: g-Deliver messages in $NCg\_Deliver^k$ in any order
26: g-Deliver messages in $Cg\_Deliver^k$ using some deterministic order
27: $G\_delivered \leftarrow (localNCg\_Deliver^k \cup NCg\_Deliver^k \cup Cg\_Deliver^k) \cup G\_delivered$
28: $k \leftarrow k + 1$
29: $pending^k \leftarrow \emptyset$
30: $localNCg\_Deliver^k \leftarrow \emptyset$

31: **when** receive$(\underline{k}, pending_q^k, \underline{ACK})$ from q
32: *#Define* $ackPSet^k(m) = \{q : p$ received $(\underline{k}, pending_q^k, \underline{ACK})$ from q **and** $m \in pending_q^k\}$
33: $ackMSet^k \leftarrow \{m : |ackPSet^k(m)| \geq n_{ack}\}$
34: $localNCmsgSet^k \leftarrow ackMSet^k \setminus (G\_delivered \cup NCmsgSet^k)$
35: $localNCg\_Deliver^k \leftarrow localNCg\_Deliver^k \cup localNCmsgSet^k$
36: g-Deliver all messages in $localNCmsgSet^k$ in any order

PROOF: The proof is by contradiction. Assume that there are two messages m and m' that conflict and are g-Delivered in stage k before Consensus. Without lack of generality, consider that m is g-Delivered by process p, and m' is g-Delivered by process q. From the Generic Broadcast algorithm (lines 31-36), p (and q) has received n_{ack} messages of the type $(k, pending^k, ACK)$ such that $m \in pending^k$ ($m' \in pending^k$). Since $n_{ack} > (n+1)/2$, there must be a process r that sends the message $(k, pending_r^k, ACK)$ to processes p and q, such that m and m' are in $pending_r^k$, contradicting Lemma 4.1. □

Lemma 4.3 relates (1) the set $Ack^k(m)$ of processes that send an acknowledgement for some message m in stage k and (2) the set Chk_p^k of processes from which some process p receives CHK messages in stage k, with (3) the set $Chk_p^k(m)$ of processes from which p receives a CHK message containing m in stage k.

Lemma 4.3 *Let $Ack^k(m)$ be a set of processes that execute the statement send(k, $pending^k, ACK$) in stage k with $m \in pending^k$, and let Chk_p^k be the set of processes from which some process p receives messages of the type $(k, pending^k, CHK)$ in stage k. If $|Ack^k(m)| \geq n_{ack}, |Chk_p^k| = n_{chk}$, and $2n_{ack} + n_{chk} \geq 2n + 1$, then there are at least $\lceil (n_{chk}+1)/2 \rceil$ processes in $Chk_p^k(m) \stackrel{def}{=} Chk_p^k \cap Ack^k(m)$.*

PROOF: We first determine a relation between sets $Ack^k(m)$, Chk_p^k, and $Chk_p^k(m)$. Set $Chk_p^k(m)$ contains all processes from set Chk_p^k that sent an acknowledgement message for m. Thus, process p knows that every process $q \in Chk_p^k(m)$ executed the statement $send(k, pending^k, ACK)$ in stage k with $m \in pending^k$, but p does not know anything about the remaining processes in $\Pi \setminus Chk_p^k$. Therefore, there are $|\Pi \setminus Chk_p^k| = (n - n_{chk})$ additional processes that might have sent an acknowledgement for m. We conclude that $|Ack^k(m)| \leq (n - n_{chk}) + |Chk_p^k(m)|$. By the hypothesis, $|Ack^k(m)| \geq n_{ack}$, and thus, $n_{ack} \leq (n - n_{chk}) + |Chk_p^k(m)|$ (1). Subtracting n from both sides in (1) leads to $n_{ack} - n \leq |Chk_p^k(m)| - n_{chk}$ (2). By rearranging $2n_{ack} + n_{chk} \geq 2n + 1$, we have that $n_{ack} - n \geq (1 - n_{chk})/2$ (3). From (2) and (3), $|Chk_p^k(m)| - n_{chk} \geq (1 - n_{chk})/2$, and so, $|Chk_p^k(m)| \geq (n_{chk} + 1)/2$. Since n_{chk} and $|Chk_p^k(m)| \in \mathbb{N}$, we conclude that $|Chk_p^k(m)| \geq \lceil (n_{chk}+1)/2 \rceil$. □

Lemma 4.4 proves property (b) of page 57. It states that any message g-Delivered by some process q during stage k, before q executes Consensus in stage k will be included in the set $NCmsgSet^k$ decided by Consensus k.

Lemma 4.4 *For any two processes p and q, and all $k \geq 1$, if process p executes the statement decide(k, $(NCmsgSet^k, -)$), then $localNCg\_Deliver_q^k \subseteq NCmsgSet^k$.*

PROOF: Let m be a message in $localNCg\_Deliver_q^k$. We first show that if p executes the statement $propose(k, majMSet_p^k, -))$, then $m \in majMSet_p^k$. Since $m \in localNCg\_Deliver_q^k$, q must have received n_{ack} messages of the type $(k, pending^k, ACK)$ (line 31) such that $m \in pending^k$. Thus, there are n_{ack} processes that sent

m to all processes in the *send* statement at line 15. From Lemma 4.3, $Chk^k(m) \geq (n_{chk}+1)/2$, and so, from the algorithm line 20, $m \in majMSet_p^k$. Therefore, for every process q that executes *propose*$(k,(majMSet_q^k,-))$, $m \in majMSet_q^k$. Let $(NCmsgSet^k,-)$ be the value decided on Consensus execution k. By the uniform validity of Consensus, there is a process r that executed *propose*$(k,(majMSet_r^k,-))$ such that $NCmsgSet^k = majMSet_r^k$, and so, $m \in NCmsgSet^k$. □

Lemma 4.5 proves property (c) of page 57.

Lemma 4.5 *If two messages m and m' conflict, then at most one of them is in $NCmsgSet^k$.*

PROOF: The proof is by contradiction. Assume that there are two messages m and m' that conflict, and are both in $NCmsgSet^k$. From the validity property of Consensus, there must be a process p that executes *propose*$(k,(majMSet_p^k,-))$, such that $NCmsgSet^k = majMSet_p^k$. Therefore, m and m' are in $majMSet_p^k$, and from the algorithm, p receives $\lceil (n_{chk}+1)/2 \rceil$ messages of the type $(k, pending^k, CHK)$ such that m is in $pending^k$, and p also receives $\lceil (n_{chk}+1)/2 \rceil$ messages of the type $(k, pending^k, CHK)$ such that m' is in $pending^k$. Since p waits for exactly n_{chk} messages of the type $(k, pending^k, CHK)$, there must exist at least one process q in Chk_p^k such that m and m' are in $pending_q^k$, contradicting Lemma 4.1. □

Lemma 4.6 lies the basis for Propositions 4.2 and 4.3. It shows that (a) if some correct process executes Consensus at some stage k, then all correct processes also execute Consensus at stage k, and (b) all correct processes g-Deliver the same messages at stage k.

Lemma 4.6 *For any two correct processes p and q, and all $k \geq 1$:*

(1) If p executes send$(k,-,CHK)$, then q eventually executes send$(k,-,CHK)$.

(2) If p executes propose$(k,-)$, then q eventually executes propose$(k,-)$.

(3) If p g-Delivers messages in $NCg\_Deliver_p^k \cup Cg\_Deliver_p^k$, then

(3.1) q also g-Delivers messages in $NCg\_Deliver_q^k \cup Cg\_Deliver_q^k$, and

(3.2) $localNCg\_Deliver_p^k \cup NCg\_Deliver_p^k =$ $localNCg\_Deliver_q^k \cup NCg\_Deliver_q^k$ and $Cg\_Deliver_p^k = Cg\_Deliver_q^k$.

PROOF: The proof is by simultaneous induction on (1), (2) and (3). (BASE STEP.) For $k=1$, we first show that (1) holds: if p executes *send*$(1,-,CHK)$ (line 17), then q also executes *send*$(1,-,CHK)$. If p executes *send*$(1,-,CHK)$, then p has R-delivered two messages, m and m', that conflict. From the agreement of R-broadcast, q also R-delivers m and m'. Assume that q first R-delivers m, and then m'. Thus, there is a time after q R-delivers m' when m and m' are in $R\_delivered_q \setminus$

$G\_delivered_q$, and $m' \notin pending_q^1$. So, q eventually executes $send(1, -, CHK)$ (line 17).

To prove (2), assume that p executes $propose(1, -)$. From the algorithm, it is clear that p executes $send(1, -, CHK)$, and from item (1) above, q also executes $send(1, -, CHK)$ and waits for n_{chk} messages of the type $(1, -, CHK)$ (line 18). Since there are n_{chk} processes correct that execute $send(1, -, CHK)$, q eventually receives n_{chk} messages of the type $(1, -, CHK)$ (line 18), and executes $propose(1, -)$.

To prove (3.1), assume that p g-Delivers messages in $NCg\_Deliver_p^1 \cup Cg\_Deliver_p^1$. Before executing $decide(1, (NCmsgSet_p^1, CmsgSet_p^1))$, p executes $propose(1, -)$. By item (2) of the lemma, q also executes $propose(1, -)$. By termination and uniform integrity of Consensus, q eventually executes $decide(1, -)$ and does it exactly once. It follows from the algorithm (lines 23-26) that q g-Delivers messages in $NCg\_Deliver_q^1 \cup Cg\_Deliver_q^1$.

To prove (3.2) we show that (a) $localNCg\_Deliver_p^1 \cup NCg\_Deliver_p^1 = localNCg\_Deliver_q^1 \cup NCg\_Deliver_q^1$, and (b) $Cg\_Deliver_p^1 = Cg\_Deliver_q^1$.

(a) From the algorithm, line 23, and the fact that initially $G\_delivered_p = \emptyset$, we have that $NCg\_Deliver_p^1 = (NCmsgSet_p^1 \setminus localNCg\_Deliver_p^1)$, and thus, $localNCg\_Deliver_p^1 \cup NCg\_Deliver_p^1 = localNCg\_Deliver_p^1 \cup (NCmsgSet_p^1 \setminus localNCg\_Deliver_p^1)$. From Lemma 4.4, it follows that $localNCg\_Deliver_p^1 \cup NCg\_Deliver_p^1 = NCmsgSet_p^1$. A similar argument follows for q, and by the agreement property of Consensus, we have $NCmsgSet_p^1 = NCmsgSet_q^1$. Therefore, we conclude that $localNCg\_Deliver_p^1 \cup NCg\_Deliver_p^1 = localNCg\_Deliver_q^1 \cup NCg\_Deliver_q^1$.

(b) From the algorithm, line 24, $Cg\_Deliver_p^1 = CmsgSet_p^1 \setminus G\_delivered_p$. Since initially $G\_delivered_p$ and $G\_delivered_q$ are empty, $Cg\_Deliver_p^1 = CmsgSet_p^1$, and $Cg\_Deliver_q^1 = CmsgSet_q^1$. By agreement of Consensus, for every p and q, $CmsgSet_p^1 = CmsgSet_q^1$, and so, $Cg\_Deliver_p^1 = Cg\_Deliver_q^1$.

(INDUCTIVE STEP.) Assume that the Lemma holds for all $k, 1 \leq k < l$. We proceed by first showing that (1) if p executes $send(l, -, CHK)$ (line 17), then q also executes $send(l, -, CHK)$. If p executes $send(l, -, CHK)$, then from line 13, there is some time t when two conflicting messages m and m' are in $R\_delivered_p \setminus G\_delivered_p$. Since m and m' are not in $G\_delivered_p$, m and m' are not in $\cup_{i=1}^{k}(localNCg\_Deliver_p^i \cup NCg\_Deliver_p^i \cup Cg\_Deliver_p^i)$. By the induction hypothesis, m and $m' \notin \cup_{i=1}^{k}(localNCg\_Deliver_q^i \cup NCg\_Deliver_q^i \cup Cg\_Deliver_q^i)$. By the agreement property of R-broadcast, eventually m and m' belong to $R\_delivered_q$. From Lemma 4.1, and the fact that m and m' conflict, there is a time after which q g-Delivers all messages in $\cup_{i=1}^{k}(localNCg\_Deliver_q^i \cup NCg\_Deliver_q^i \cup Cg\_Deliver_q^i)$ such that there exist two messages m and m' in $R\_delivered_q \setminus G\_delivered_q$, and m and m' are not both in $pending_q^l$. Thus, q eventually executes $send(l, -, CHK)$.

Suppose that (2) p executes $propose(l, -)$. From the algorithm, p previously executed $send(l, -, CHK)$, and from item (1), q also executes $send(l, -, CHK)$ and

waits for n_{chk} messages of the type $(l, -, CHK)$. Since there are n_{chk} processes correct that execute $send(l, -, CHK)$, q eventually receives n_{chk} messages of the type $(l, -, CHK)$, and executes $propose(l, -)$.

We now consider that (3.1) p g-Delivers messages in $NCg\_Deliver_p^l \cup Cg\_Deliver_p^l$. Before executing $decide(l, (NCmsgSet_p^l, CmsgSet_p^l))$, p executes $propose(l, -)$. By item (1) of the lemma, q also executes $propose(l, -)$. By the termination and agreement properties of Consensus, q eventually executes $decide(l, -)$ exactly once. From the algorithm, q g-Delivers messages in $NCg\_Deliver_q^l \cup Cg\_Deliver_q^l$.

To prove (3.2) we show that (a) $localNCg\_Deliver_p^l \cup NCg\_Deliver_p^l = localNCg\_Deliver_p^l \cup NCg\_Deliver_q^l$, and (b) $Cg\_Deliver_p^l = Cg\_Deliver_q^l$.

(a) From the algorithm, line 23, and Lemma 4.4 (i.e., $localNCg\_Deliver_p^l \subseteq NCmsgSet_p^l$), it follows that $localNCg\_Deliver_p^l \cup NCg\_Deliver_p^l = NCmsgSet_p^l - G\_delivered_p$. To see why, note that from lines 34 and 35, $localNCg\_Deliver_p^l \cap G\_delivered_p = \emptyset$. By the agreement property of Consensus, $NCmsgSet_p^l = NCmsgSet_q^l$. From the algorithm, $G\_delivered = \cup_{i=1}^{k}(localNCg\_Deliver^i \cup NCg\_Deliver^i \cup Cg\_Deliver^i)$, and from the induction hypothesis, $G\_delivered_p = G\_delivered_q$. Therefore, we have $localNCg\_Deliver_p^l \cup NCg\_Deliver_p^l = localNCg\_Deliver_p^l \cup NCg\_Deliver_q^l$.

(b) From the algorithm, line 24, $Cg\_Deliver_p^l = CmsgSet_p^l \setminus G\_delivered_p$. But when line 24 is evaluated, $G\_delivered_p = \cup_{i=1}^{k}(localNCg\_Deliver_p^i \cup NCg\_Deliver_p^i \cup Cg\_Deliver_p^i)$, and it follows from the induction hypothesis that $G\_delivered_p = G\_delivered_q$. By the agreement property of Consensus, $CmsgSet_p^l = CmsgSet_q^l$, and thus, $Cg\_Deliver_p^l = Cg\_Deliver_q^l$. □

The following propositions suppose $f < \max(n_{ack}, n_{chk})$. Proposition 4.2 is stronger than the agreement property defined in Section 4.1, since it claims that any two correct processes not only g-Deliver the same messages, but also g-Deliver them in the same stage.

The proof for Proposition 4.2 considers two cases. The first case assumes that some correct process p g-Delivers m in stage k and executes Consensus in stage k. In this case, $m \in localNCg\_Deliver_p^k \cup NCg\_Deliver_p^k \cup Cg\_Deliver_p^k$, and from Lemma 4.6, every correct process also g-Delivers m. The second case considers that p g-Delivers m in stage k but never executes Consensus in stage k. The proof proceeds by showing that if this happens, then all correct processes send an acknowledgement for m in stage k, and eventually every correct process receives n_{ack} acknowledgement messages for m and g-Delivers m.

Proposition 4.2 (AGREEMENT). *If a correct process p g-Delivers a message m in some stage k, then every correct process q eventually g-Delivers m in stage k.*

PROOF: There are two cases to consider: (a) p executes Consensus on stage k, and (b) p never executes Consensus in stage k.

(a) Since p g-Delivers m in stage k, $m \in localNCg\_Deliver_p^k \cup NCg\_Deliver_p^k \cup Cg\_Deliver_p^k$, and so, from Lemma 4.6, we have $m \in localNCg\_Deliver_q^k \cup NCg\_Deliver_q^k \cup Cg\_Deliver_q^k$. Thus, q either g-Delivers m at line 36 (in which case $m \in localNCg\_Deliver_q^k$), or at line 25 (in which case $m \in NCg\_Deliver_q^k$), or at line 26 (in which case $m \in Cg\_Deliver_q^k$).

(b) Since p does not execute Consensus in stage k, $m \in localNCg\_Deliver_p^k$, and it must be that p has received n_{ack} messages of the type $(k, pending^k, ACK)$ (line 30) such that $m \in pending^k$. There are $n_{ack} \geq (n+1)/2$ correct processes, and so, p has received the message $(k, pending^k, ACK)$ from at least one correct process r.

We claim that every correct process r' executes the $send(k, pending^k, ACK)$ statement at line 15, such that $m \in pending^k$. From lines 12-15, r R-delivers m, and by the agreement of Reliable Broadcast, eventually r' also R-delivers m. Therefore, there is a time t when $m \in R\_delivered_{r'}$.

It follows from the fact that m is g-Delivered by p in stage k that $m \notin \cup_{i=1}^{k-1}(localNCg\_Deliver_p^k \cup NCg\_Deliver_p^k \cup Cg\_Deliver_p^k)$. By Lemma 4.6, we have $m \notin \cup_{i=1}^{k-1}(localNCg\_Deliver_{r'}^k \cup NCg\_Deliver_{r'}^k \cup Cg\_Deliver_{r'}^k)$, and so, there is a $t' > t$ when r' executes line 13. At time t', m does not conflict with any other message. To see why, consider that m conflicts with some message m' in stage k. In this case, r' executes $send(k, -, CHK)$ in stage k, and from Lemma 4.6 all correct processes also execute $send(k, -, CHK)$ in stage k. It follows that r' eventually executes Consensus in stage k, a contradiction that concludes the claim.

Since there are n_{ack} correct processes that execute $send(k, pending^k, ACK)$, such that $m \in pending^k$, q will eventually execute the *when* statement at line 31, and g-Deliver m. □

Two situations are distinguished in the proof for Proposition 4.3. The first situation (a) considers that some process q g-Delivers two conflicting messages m and m' in the same stage k, and in the second situation, (b) process q g-Delivers m and m' in different stages. Considering that q g-Delivers m before m', it is shown for (a) that since m and m' conflict, $m' \in CmsgSet^k$. Assuming, for a contradiction, that p g-Delivers m' before m, from a similar argument, it is concluded that $m \in CmsgSet^k$. Therefore, $m, m' \in CmsgSet^k$. However, all messages in $CmsgSet^k$ are g-Delivered in the same deterministic order, and thus, it cannot be that q g-Delivers first m and then m', and p g-Delivers first m' and then m. For situation (b), it follows directly from Algorithm 1 that if p and q both g-Deliver m, respectively m', in stage k, respectively k', then they g-Deliver m and m' in the same order.

Proposition 4.3 (Partial Order). *If correct processes p and q both g-Deliver messages m and m', and m and m' conflict, then p g-Delivers m before m' if and only if q g-Delivers m before m'.*

Proof: Assume that q g-Delivers message m before message m'. We show that p also g-Delivers m before m'. There are two cases to consider: (a) q g-Delivers m and

m' at stage k, and by Proposition 4.2, p also g-Delivers m and m' at stage k, and (b) q g-Delivers m at stage k, and m' at stage $k' > k$, and by Proposition 4.2, p also g-Delivers m at stage k, and m' at stage k'.

(a) We claim that if messages m and m' conflict, and q g-Delivers m before m', then $m' \in CmsgSet^k$. To see why, notice that if q g-Delivers m before Consensus, $m \in localNCmsgSet_q^k$, and from Lemma 4.4, $m \in NCmsgSet^k$. From Lemma 4.5, m and m' cannot be both in $NCmsgSet^k$, thus $m' \in CmsgSet^k$, concluding the proof of our claim.

Suppose, by way of contradiction, that p g-Delivers m' before m. From an argument similar to the claim above, $m \in CmsgSet^k$. Therefore, m and m' are in $CmsgSet^k$, and m and m' are g-Delivered by q and p at line 26. However, since q g-Delivers m before m', and messages in $CmsgSet^k$ are g-Delivered according to some deterministic function, p and q do not use the same deterministic function. A contradiction that concludes the proof of case (a).

(b) From the algorithm, if p g-Delivers m at stage k, and m' at stage $k' > k$, then p g-Delivers m before m'. □

Proposition 4.4 below proves that Algorithm 1 guarantees the validity property of Generic Broadcast using two mechanisms [CT96]. The idea is to assume that some message m is g-Broadcast and never g-Delivered and then reach a contradiction. First, the proof of Proposition 4.4 shows that if m is never g-Delivered, then there is a Consensus execution k_1 when every correct process proposes m. Notice that this can only be proved for correct processes, since to propose m, a process first has to R-deliver m, and the properties of Reliable Broadcast only ensure that correct processes R-deliver all R-broadcast messages. It then shows that from the definition of faulty processes, there is a Consensus execution k_2 that no faulty process executes (they all crash before k_2). The contradiction follows immediately, since at Consensus execution $k = \max(k_1, k_2)$, only correct processes propose a value, and m is always proposed. Thus, m is included in the decision of Consensus k, and will be g-Delivered.

Proposition 4.4 (VALIDITY). *If a correct process p g-Broadcasts a message m, then p eventually g-Delivers m.*

PROOF: For a contradiction, assume that p g-Broadcasts m but never g-Delivers it. From Proposition 4.2, no correct process g-Delivers m. Since p g-Broadcasts m, it R-broadcasts m, and from the validity property of Reliable Broadcast, p eventually R-delivers m. By Algorithm 1, there is a time after which $m \in R\_delivered_p$. It follows from the agreement property of Reliable Broadcast and the fact that m is never g-Delivered, that eventually, for every correct process q, $m \in (R\_delivered_q \setminus G\_delivered_q)$.

By the contradiction hypothesis, p does not g-Deliver m, and so, p does not receive n_{ack} messages of the type $(k, pending^k, ACK)$ such that $m \in pending^k$. But since there are n_{ack} correct processes that execute the *if* statement at line 13, there is at least one correct process q such that, after $m \in R\_delivered_q \setminus G\_delivered_q$, q never

executes the *then* branch (lines 14-15), and always executes the *else* branch (lines 17-30). Thus, q executes *send*$(k, -, CHK)$. From Lemma 4.6, item (1), every correct process also executes *send*$(k, -, CHK)$. Since there are n_{chk} correct processes, no correct process remains blocked forever at the *wait* statement (line 18), and every correct process eventually executes *propose*$(k, -)$. Thus, there is a k_1 such that for all $l \geq k_1$, all correct processes execute *propose*$(l, (majMSet^l, (R\_delivered \setminus G\_delivered) \setminus majMSet^l))$, and $m \in majMSet^l \cup (R\_delivered \setminus G\_delivered)$.

Assume that k_2 is such that no faulty process executes *propose*$(l, -)$, $l \geq k_2$, (i.e., at k_2 all faulty processes have crashed). Let $k = \max(k_1, k_2)$. All correct processes execute *propose*$(k, -)$, and by the termination and agreement of Consensus, all correct processes execute *decide*$(k, (NCmsgSet^k, CmsgSet^k))$ with the same $(NCmsgSet^k, CmsgSet^k)$. By the uniform validity property of Consensus, some process q executes *propose*$(k, (majMSet^l, (R\_delivered \setminus G\_delivered) \setminus majMSet^l))$ such that $m \in majMSet^l \cup (R\_delivered \setminus G\_delivered)$, and so, all processes g-Deliver m, a contradiction that concludes the proof. □

Proposition 4.5 (UNIFORM INTEGRITY). *For any message m, each process g-Delivers m at most once, and only if m was previously g-Broadcast by sender(m).*

PROOF: If a process p g-Delivers m at line 36, then p received n_{ack} messages of the type $(k, pending^k, ACK), m \in pending^k$. Let q be a process from which p received the message $(k, pending^k_q, ACK), m \in pending^k_q$. Since q executes *send*$(k, pending^k_q, ACK)$, q has R-delivered m. By the uniform integrity of Reliable Broadcast, process *sender(m)* has R-broadcast m, and so, *sender(m)* has g-Broadcast m.

Now consider that p g-Delivers m at line 25 or 26. Thus, p executed the statement *decide*$(k, (NCmsgSet^k, CmsgSet^k))$ for some k, such that $m \in NCmsgSet^k \cup CmsgSet^k$. By the uniform validity property of Consensus, some process q must have executed *propose*$(k, (majMSet^k, (R\_delivered \setminus G\_delivered) \setminus majMSet^k))$ such that $m \in majMSet^k \cup (R\_delivered \setminus G\_delivered)$. We distinguish two cases.

Case (a). If $m \in majMSet^k_q$, then, from Algorithm 1, $|Chk^k_q(m)| \geq \lceil (n_{chk} + 1)/2 \rceil$. Let $r \in Chk^k_q(m)$. Therefore, r executed *send*$(k, pending^k_r, CHK)$, such that $m \in pending^k_r$, and thus, r has R-delivered m.

Case (b). If $m \in R\_delivered \setminus G\_delivered$, it is not difficult to see that q has R-delivered m.

In both cases, by the uniform integrity property of Reliable Broadcast, process *sender(m)* has R-broadcast m, and so, *sender(m)* has g-Broadcast m. □

Theorem 4.1 *Algorithm 1 solves Generic Broadcast, or reduces Generic Broadcast to a sequence of Consensus in asynchronous systems with $f < max(n_{ack}, n_{chk})$.*

PROOF. Immediate from Propositions 4.2, 4.3, 4.4, and 4.5. □

4.3 Evaluation of the Generic Broadcast Algorithm

The Generic Broadcast algorithm is strict and cheaper than known Atomic Broadcast implementations based on the same assumptions. We evaluate next the cost of the Generic Broadcast algorithm using the delivery latency parameter, introduced in this section.

4.3.1 Generic Broadcast Algorithm Strictness

Proposition 4.6 states that the Generic Broadcast algorithm of Section 4.2.2 is a strict implementation of Generic Broadcast.

Proposition 4.6 *Algorithm 1 is a strict Generic Broadcast algorithm.*

PROOF. Immediate from Figure 4.1, where process p g-Broadcasts message m and process s g-Broadcasts messages m'. Process p (respectively s) R-delivers m' (respectively m) after g-Delivering m' (respectively m) – not shown in Figure 4.1. Process p only acknowledges message m, processes q and r acknowledge messages m and m', and process s only acknowledges message m'.

Process p receives the acknowledges from p, q, and r and since $n_{ack} = 3$, p g-Delivers m. Process p then receives the acknowledgement from s for m', and g-Delivers m'. Similarly, s g-Delivers m' and then m. Therefore, p g-Delivers m before m', and s g-Delivers m' before m. □

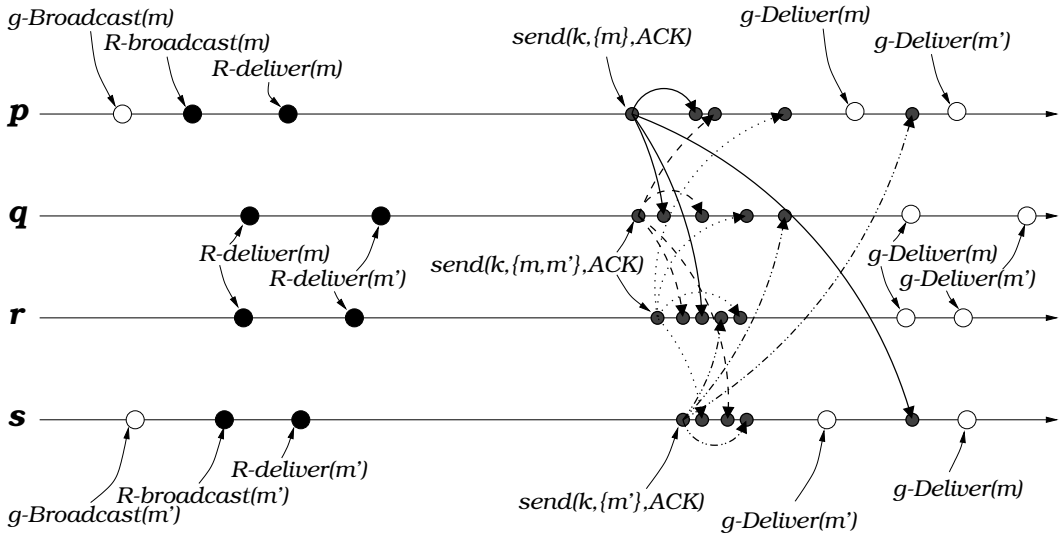

Figure 4.1: Run R of Generic Broadcast ($n_{ack} = 3$)

4.3.2 Generic Broadcast Algorithm Cost

In order to analyse the cost of the Generic Broadcast algorithm, we introduce the delivery latency parameter. We analyse the Generic Broadcast algorithm considering best case runs, when messages can be g-Delivered without conflict, and with conflict.

Delivery Latency. In the following, we introduce the delivery latency as a parameter to measure the efficiency of algorithms solving any Broadcast problem (defined by the primitives α-Broadcast and α-Deliver). The delivery latency is a variation of the Latency Degree introduced in [Sch97], which is based on modified Lamport's clocks [Lam78]:

- a *send* event and a *local* event on a process p do not modify p's local clock,
- let $ts(send(m))$ be the timestamp of the $send(m)$ event, and $ts(m)$ the timestamp carried by message m: $ts(m) \stackrel{\text{def}}{=} ts(send(m)) + 1$, and
- the timestamp of $receive(m)$ on a process p is the maximum between $ts(m)$ and p's current clock value.

The delivery latency of a message m α-Broadcast in some run R of an algorithm A solving a Broadcast problem, denoted $dl^R(m)$, is defined as the difference between (1) the largest timestamp of all α-Deliver(m) events (at most one per process) in run R and (2) the timestamp of the α-Broadcast(m) event in run R. Let π_m^R be the set of processes that α-Deliver message m in run R. The delivery latency of m in R is formally defined as

$$dl^R(m) \stackrel{\text{def}}{=} \max_{p \in \pi_m^R} (ts(\alpha\text{-Deliver}_p(m)) - ts(\alpha\text{-Broadcast}(m))).$$

For example, consider a broadcast algorithm A_b where (1) to broadcast a message m, a process p sends m to all processes, (2) each process q on receiving m sends an acknowledgement message $ACK(m)$ to all processes, and (3) as soon as q has received n_{ack} messages of the type $ACK(m)$, q delivers m. Let R be a run of A_b, as shown in Figure 4.2. In this case we have $dl^R(m) = 2$.

The delivery latency is a measure of the synchronisation among processes in a given run produced by some broadcast algorithm A to deliver a message. The delivery latency can be used to characterise the *minimal synchronisation* among processes, required by an algorithm A, to deliver messages. For example, algorithm A_b requires that processes send an $ACK(m)$ message only after receiving message m, and so, no run generated by A_b where m is broadcast will have $send_p(ACK(m))$ preceding $receive_p(m)$, for any process p. Nevertheless, algorithm A_b allows a process q to send $ACK(m)$ after having received $ACK(m)$ from some process p. Thus, there exists a run R' of A_b where $receive_q(ACK(m))$ precedes $send_q(ACK(m))$ (see Figure 4.3). In this case we have $dl^{R'}(m) = 3$.

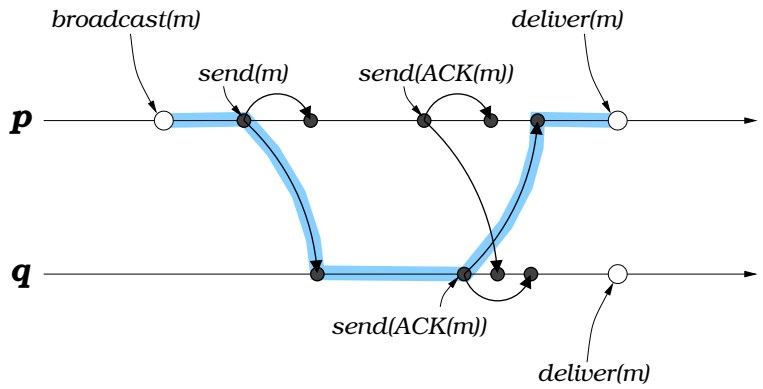

Figure 4.2: Run R of A_b with $dl^R(m) = 2$

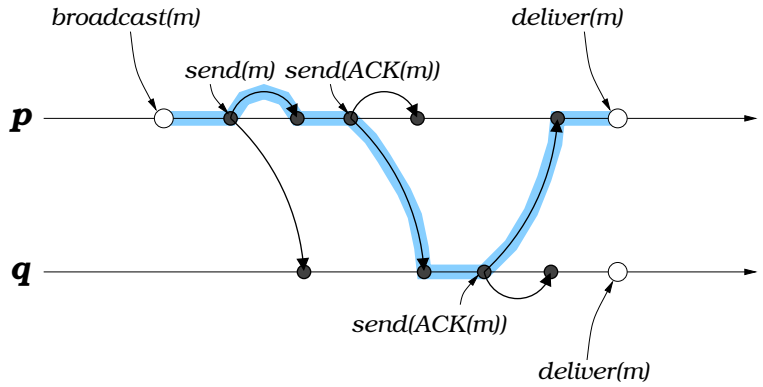

Figure 4.3: Run R' of A_b with $dl^{R'}(m) = 3$

Therefore, when characterising a broadcast algorithm A with the delivery latency parameter, we will consider best case scenarios, which characterise the minimal synchronisation necessary to deliver messages.

Cost Analysis. We now discuss the cost of our Generic Broadcast algorithm. Our main result is that for messages that do not conflict, the Generic Broadcast algorithm can deliver messages with a delivery latency equal to 2, while for messages that conflict, the delivery latency is at least equal to 4. Since known Atomic Broadcast algorithms deliver messages with a delivery latency of at least 3,[5] this results shows the tradeoff of the Generic Broadcast algorithm: if messages conflict frequently, our Generic Broadcast algorithm may become less efficient than an Atomic Broadcast algorithm, while if conflicts are rare, then our Generic Broadcast algorithm leads to smaller costs compared to Atomic Broadcast algorithms.

Before stating Properties 4.8 – 4.11, we present Proposition 4.7 which defines a lower bound on algorithms that implement Reliable Broadcast, and Corollary 4.1 which

[5] An exception is the Optimistic Atomic Broadcast algorithm (see Chapter 5), which can deliver messages with delivery latency equal to 2 if the *spontaneous total order property* holds.

states this lower bound in terms of the delivery latency parameter.

Proposition 4.7 *Let A_{rb} be an algorithm that solves Reliable Broadcast. For every run R of A_{rb} where a process p R-broadcasts some message m and a process $q \neq p$ R-delivers m, there is a causal chain of events connecting R-broadcast$_p(m)$ and R-deliver$_q(m)$.*

Proof. Suppose, by way of contradiction, that there exists an algorithm A_{rb} that solves Reliable Broadcast such that in some runs of A_{rb}, a process p R-broadcasts a message m, a process $q \neq p$ R-delivers m, and there is no causal chain of events connecting *R-broadcast*$_p(m)$ and *R-deliver*$_q(m)$. Let R be such a run of A_{rb} where *R-broadcast*$_p(m)$ is the first event executed by process p. From the hypothesis, there is no event $e \in R$ so that *R-broadcast*$_p(m) \rightarrow e$ and $e \rightarrow$ *R-deliver*$_q(m)$.

Consider now a run R' similar to R except that p does not R-broadcast m. Process q is not able to distinguish between R and R', and since q R-delivers m in R, q R-delivers m in R', violating the uniform integrity property of Reliable Broadcast, and contradicting our hypothesis that A_{rb} solves Reliable Broadcast. □

Corollary 4.1 *There is no algorithm A_{rb} that implements Reliable Broadcast such that for any message m R-delivered in some run R produced by A_{rb}, $dl^R(m) < 1$.*

Proof. Immediate from Proposition 4.7 and the definition of delivery latency. □

Propositions 4.8 and 4.9 assess the cost of the Generic Broadcast algorithm when messages do not conflict. Proposition 4.8 defines a lower bound on the delivery latency of Algorithm 1 for messages g-Delivered without Consensus (line 36), and Proposition 4.9 shows that this bound can be reached in runs where there are no process failures.

Proposition 4.8 *There is no run R generated by Algorithm 1 where some message m is only g-Delivered at line 36 and $dl^R(m) < 2$.*

Proof. Assume for a contradiction that there is a run R and a message m g-Delivered in R such that $dl^R(m) < 2$. Since m is g-Delivered in R, by the integrity property of Generic Broadcast, there is a process q that g-Broadcasts m. By Algorithm 1, if q g-Broadcasts m, q R-broadcasts m, and every process p that g-Delivers m first R-delivers m. We define $l^{RB}_{m,p} = ts(\textit{R-deliver}_p(m)) - ts(\textit{R-broadcast}_q(m))$, and $l^{gB}_{m,p} = ts(\textit{g-Deliver}_p(m)) - ts(\textit{g-Broadcast}_q(m))$, where $l^{gB}_{m,p} \geq l^{RB}_{m,p}$. By Corollary 4.1, $l^{RB}_{m,p} \geq 1$, and from the definition of delivery latency and the contradiction hypothesis, $l^{gB}_{m,p} \leq dl^R(m) < 2$.

It follows that $1 \leq l^{RB}_{m,p} \leq l^{gB}_{m,p} \leq dl^R(m) < 2$, and it must be that $l^{RB}_{m,p} = l^{gB}_{m,p}$. Therefore, after R-delivering m, p does not receive any message m' such that $m \rightarrow m'$, where $\rightarrow$ is the happens-before relation defined by Lamport [Lam78]. Since p g-Delivers m at line 36, p receives n_{ack} messages of the type $(k, pending^k, ACK)$,

such that $m \in pending^k$. Let r be a process from which p receives a message $(k, pending^k, ACK)_r$ at line 18. Since $m \in pending^k$, than r has received m, and so, $m \rightarrow (k, pending^k, ACK)_r$, a contradiction. □

Proposition 4.9 *Assume that Algorithm 1 uses the Reliable Broadcast implementation given in [CT96]. There is a run R generated by Algorithm 1 where message m is g-Delivered at line 36 and $dl^R(m) = 2$.*

PROOF. Immediate from Figure 4.4 where process p g-Broadcasts a message m. (Some messages have been omitted from Figure 4.4 for clarity.) For all $\rho \in \{p, q, r, s\}$, $ts(receive_\rho(m)) = ts(send_p(m)) + 1$, and, for all $\rho' \in \{p, q, s\}$, $ts(receive_\rho(k, \{m\}, ACK)$ from $\rho') = ts(send_{\rho'}(k, \{m\}, ACK)) + 1$. But $ts(send_{\rho'}(k, \{m\}, ACK)) = ts(receive_{\rho'}(m))$, and so, $ts(receive_\rho(k, \{m\}, ACK)$ from $\rho') = ts(send_p(m)) + 2$. From Figure 4.4, $ts(g\text{-}Broadcast_p(m)) = ts(send_p(m))$, and $ts(g\text{-}Deliver_\rho(m)) = ts(receive_\rho(k, \{m\}, ACK)$ from $\rho')$. By the definition of delivery latency, we have $dl^R(m) = 2$. □

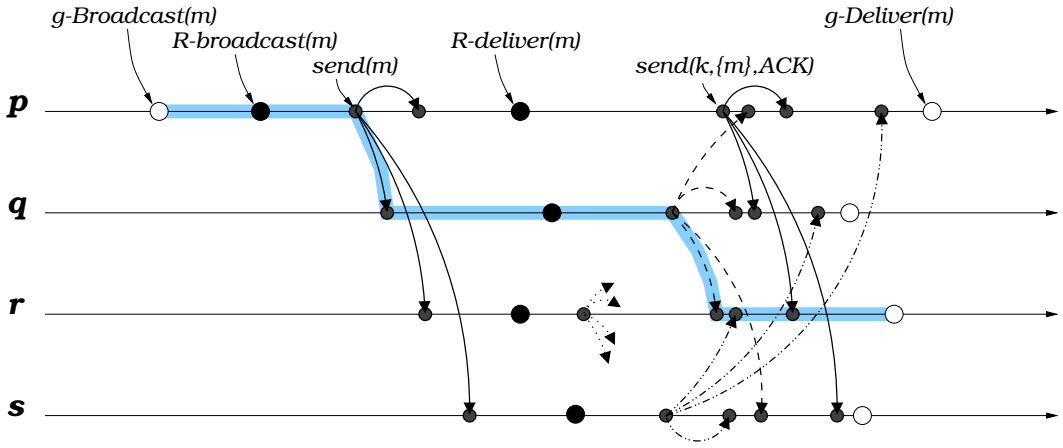

Figure 4.4: Run of Generic Broadcast with $dl^R(m) = 2$

The results that follow are about the cost of the Generic Broadcast algorithm in runs where conflicting messages are g-Broadcast. Proposition 4.10 establishes a lower bound for cases where messages conflict, and Proposition 4.11 shows that the best case with conflicts can be reached in runs with no process failures nor failure suspicions. Proposition 4.10 is based on Conjecture 4.1, which establishes a lower bound on Consensus algorithms. This lower bound is based on the latency degree parameter, introduced in [Sch97].

Conjecture 4.1 *Assume an asynchronous system model M augmented with a failure detector that does not satisfy strong accuracy. There is no algorithm A_C in M that solves Consensus with a latency degree smaller than 2.*

Notice that there are algorithms that solve Consensus with a latency degree equal to 2 in M [Sch97, MR99].

Proposition 4.10 *Assume that Conjecture 4.1 is true. There is no run R generated by Algorithm 1 in M where m and m' are the only messages g-Delivered, m and m' conflict, and $dl^R(m) < 4$ and $dl^R(m') < 4$.*

Proof. Assume for a contradiction that there is a run R and two messages m and m' g-Delivered in R such that m and m' conflict and $dl^R(m) < 4$ and $dl^R(m') < 4$. From Lemma 4.2, at most one message is g-Delivered in R before Consensus. Without loss of generality, assume that m is g-Delivered after Consensus. We will show that it cannot be that $dl^R(m) < 4$.

For every process p that g-Delivers m, p first executes $decide_p(-, (set_a, set_b))$, such that $m \in set_b$. Thus, from the uniform validity property of Consensus, there is a process q that executes $propose_q(-, (set_a, set_b))$. Let $l_p^C = ts(decide_p(-, (set_a, set_b))) - ts(propose_q(-, (set_a, set_b)))$. From Conjecture 4.1 and the definition of latency degree [Sch97], $l_p^C \geq 2$.

From Algorithm 1, before q executes $propose_q(-, (set_a, set_b))$, q receives n_{chk} messages of the type $(-, pending, CHK)$. Let r be a process from which q receives message $(-, pending_r, CHK)$. We claim that r R-delivers m. To see why, notice that in Algorithm 1, r only executes $send_r(-, pending_r, CHK)$ (line 17) after r R-delivers two conflicting messages: m and m'. It follows that $ts(g\text{-}Deliver_p(m)) - ts(R\text{-}deliver_r(m)) \geq l_p^C + 1$, and since $l_p^C \geq 2$, we have (a) $ts(g\text{-}Deliver_p(m)) - ts(R\text{-}deliver_r(m)) \geq 3$.

We define (b) $l_{m,r}^{RB} = ts(R\text{-}deliver_r(m)) - ts(R\text{-}broadcast(m))$, and (c) $l_{m,p}^{gB} = ts(g\text{-}Deliver_p(m)) - ts(g\text{-}Broadcast(m))$. By Algorithm 1, (d) $ts(g\text{-}Broadcast(m)) = ts(R\text{-}broadcast(m))$. It follows from (a), (b), (c), and (d) that $l_{m,p}^{gB} = l_{m,p}^{RB} + 3$.

By the contradiction hypothesis, $dl^R(m) < 4$, and by the definition of delivery latency, for all p that g-Deliver m, $l_{m,p}^{gB} \leq dl^R(m)$. Thus, $l_{m,p}^{RB} + 3 \leq dl^R(m) < 4$. We conclude that $l_{m,p}^{RB} = 0$, which contradicts Proposition 4.7 and concludes the proof.□

Proposition 4.11 *Assume that Algorithm 1 uses the Reliable Broadcast implementation given in [CT96], and the Consensus implementation given in [Sch97]. There exists a run R of Algorithm 1 where two messages conflicting m and m' are g-Delivered in some stage k, and $dl^R(m) = 4$ and $dl^R(m') = 4$.*

Proof. Immediate from Figure 4.5, where process q g-Broadcasts message m, and process r g-Broadcasts message m'. (The Consensus execution and some messages have been omitted for clarity.) For all $\rho \in \{p, q, r, s\}$, $ts(receive_\rho(m)) = ts(send_q(m)) + 1$, and $ts(receive_\rho(m')) = ts(send_r(m')) + 1$. It also follows that for all $\rho' \in \{p, q, r\}$, $ts(receive_\rho(k, -, CHK)$ from $\rho') = ts(send_{\rho'}(k, -, CHK)) + 1$. From Figure 4.5, $ts(send_{\rho'}(k, -, CHK)) = ts(receive_{\rho'}(m)) = ts(receive_{\rho'}(m'))$, and thus, $ts(receive_\rho(k, -, CHK)$ from $\rho') = ts(send_{\rho''}(m)) + 2$, $\rho'' \in \{q, r\}$.

By the Consensus algorithm given in [Sch97], $ts(decide_\rho(-)) = ts(propose_\rho(-)) + 2$. From Figure 4.5, $ts(propose_\rho(-)) = ts(receive_\rho(k, -, CHK))$, and we have that $ts(decide_\rho(-)) = ts(receive_\rho(k, -, CHK)) + 4$. We conclude by the definition of delivery latency and since $ts(g\text{-}Deliver_\rho(m)) = ts(g\text{-}Deliver_\rho(m')) = ts(decide_\rho(-))$,

$ts(g\text{-}Broadcast_q(m)) = ts(send_q(m))$, and $ts(g\text{-}Broadcast_r(m)) = ts(send_r(m))$, that $dl^R(m) = 4$ and $dl^R(m') = 4$. □

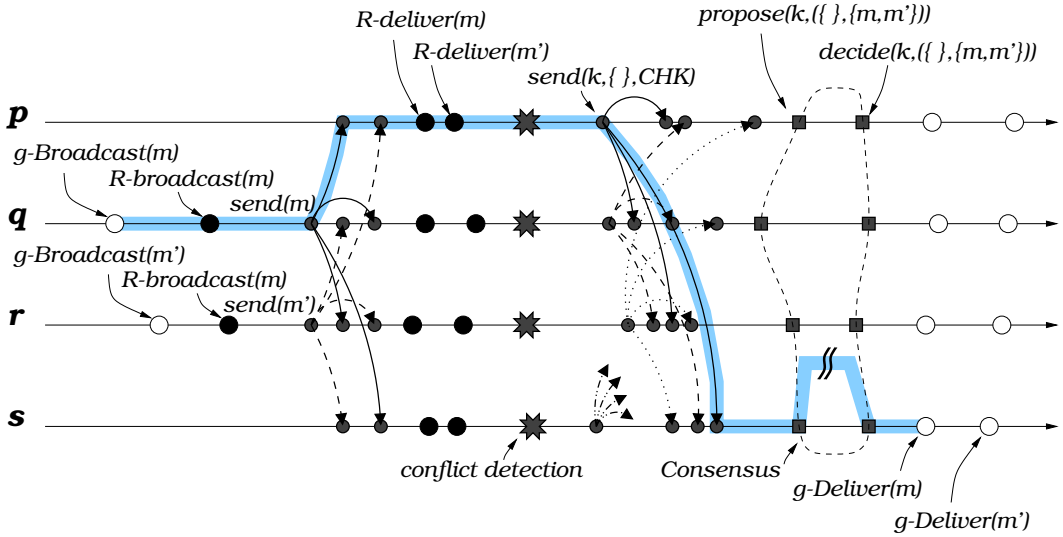

Figure 4.5: Run of Generic Broadcast with $dl^R(m) = 4$ and $dl^R(m') = 4$

4.4 Related Work

Group communication aim at extending traditional one-to-one communication, which is insufficient in many settings. One-to-many communication is typically needed to handle replication (replicated data, replicated objects, etc.). Classical techniques to manage replicated data are based on voting and quorum systems (e.g., [Gif79, Her86, JM87] to cite a few). Early quorum systems distinguish read operations from write operations in order to allow for concurrent read operations. These ideas have been extended to abstract data types in [Her86]. Increasing concurrency without compromising strong consistency guarantees on replicated data is a standard way to increase system performance. Lazy replication [RL92] is another approach that aims at increasing performance by reducing the cost of replication. Lazy replication also distinguishes between read and write operations, and relaxes the requirement of total order delivery of read operations. Consistency is ensured at the cost of managing timestamps outside the set of replicated servers. Timestamps are used to ensure Causal Order delivery on the replicated servers.

Our approach also aims at increasing the performance of replication by increasing concurrency in the context of group communication. To the best of our knowledge, no previous work has defined group communication in this way. Nevertheless, there are some similarities between our Generic Broadcast algorithm and quorum system [Gif79]. From this perspective, our work can be seen as a way to integrate group communication and quorum systems. There is even a stronger similarity between quorum systems and our Generic Broadcast algorithm. Our algorithm is based on

two sets: an acknowledgement set and a checking set. These sets play a role similar to quorum systems. However, quorum systems require weaker conditions to keep consistency than the condition required by the acknowledgement and checking sets.[6] This discrepancy is explained in part by the fact that quorum systems are only concerned with safety guarantees (e.g., two writes on replicas of the same object should not be performed concurrently), whereas the Generic Broadcast algorithm is concerned with safety *and* liveness guarantees.

4.5 Discussion

Generic Broadcast is a powerful message ordering abstraction. The definition of a Generic Broadcast primitive is based on a conflict relation derived from semantic information provided by the application. Reliable and Atomic Broadcast are special cases of Generic Broadcast, where the conflict relation is the empty set in one case (i.e., Reliable Broadcast) and the Cartesian Product over all messages in the other case (i.e., Atomic Broadcast). Reliable and Atomic Broadcast determine the two ends of a spectrum of order relations. Between these two ends, we defined a conflict relation to be used by the Database State Machine algorithm, which characterises a serialisability based message ordering.

The advantage of Generic Broadcast over Atomic Broadcast is a cost issue, where cost is defined by the delivery latency of messages. The intuition behind the Generic Broadcast problem is that ordering messages has a cost, and this cost should only be paid when necessary, that is, when messages conflict. This notion of cost is formally defined by the strictness property. In this chapter, we have presented a strict Generic Broadcast algorithm.

On a different issue, the Generic Broadcast algorithm proposed uses mechanisms that have similarities with quorum systems. This raises an interesting issue and lays the basis for further investigation aiming at better understanding the differences between replication protocols based on group communication (e.g., Atomic Broadcast, Generic Broadcast) and replication protocols based on quorum systems.

Finally, the Generic Broadcast algorithm proposed requires at least $(2n+1)/3$ correct processes. Such a condition is usual in the context of Byzantine failures, but rather surprising in the context of crash failures. These observations suggest that there might be room for optimised Generic Broadcast algorithms.

[6] Let n_r be the size of a read quorum, and n_w the size of a write quorum. Quorum systems usually requires that $n_r + n_w \geq n + 1$, and $n_w \geq \lceil (n+1)/2 \rceil$.

Chapter 5

Optimistic Atomic Broadcast

A pessimist sees the difficulty in every opportunity;
an optimist sees the opportunity in every difficulty.

Winston Churchill

Broadcast protocols have been shown to play an important role in fault tolerant systems. For replication mechanisms based on the state machine approach [Sch90] (e.g., the Database State Machine), Atomic Broadcast guarantees that every replica delivers requests in the same order. One way of improving the efficiency of such replication mechanisms is to use broadcast primitives providing order guarantees that take advantage of application semantics, like Generic Broadcast. Another way is to exploit system properties to implement fast Atomic Broadcast protocols.

In this chapter, we introduce optimistic approaches to implementing broadcast protocols (e.g., Atomic Broadcast). These approaches are optimistic because they are based on system properties that do not always hold, but if these properties hold for a certain period, messages can be delivered fast. This chapter describes three optimistic approaches in general lines, and presents one in detail, the Optimistic Atomic Broadcast (OPT-ABcast) algorithm. The optimism in these approaches exploits the spontaneous total order property, that is, the fact that in some networks it is highly probable that messages are received in the same total order.

5.1 Degrees of Optimism

The optimistic approaches presented in this chapter exploit the *spontaneous total order* property to deliver messages fast. The spontaneous total order property holds under some circumstances (e.g., moderate load) in local area networks. It can be stated as follows.

> (Spontaneous Total Order) Consider a set Ω of processes. If a process p sends a message m to all processes in Ω, and a process q sends a message m'

to all processes in Ω, then the two messages are received in the same order by all receivers.

Under abnormal execution conditions (e.g., high network loads), the spontaneous total order property may be violated. More generally, one can consider that the system passes through periods when the spontaneous total order property holds, and periods when the property does not hold.

To illustrate the spontaneous total order property, we conducted some experiments involving eight workstations (UltraSparc 1+) connected by an Ethernet network (10 Mbits/s). In the experiments (see Figure 5.1), each workstation broadcasts messages to all the other workstations, and receives messages from all workstations over a certain period of time (around 10 sec.). Broadcasts are implemented with IP-multicast, and messages have 1024 bytes. From Figure 5.1, it can be seen that there is a relation between the time between successive broadcast calls, and the percentage of messages that are received in the same order.

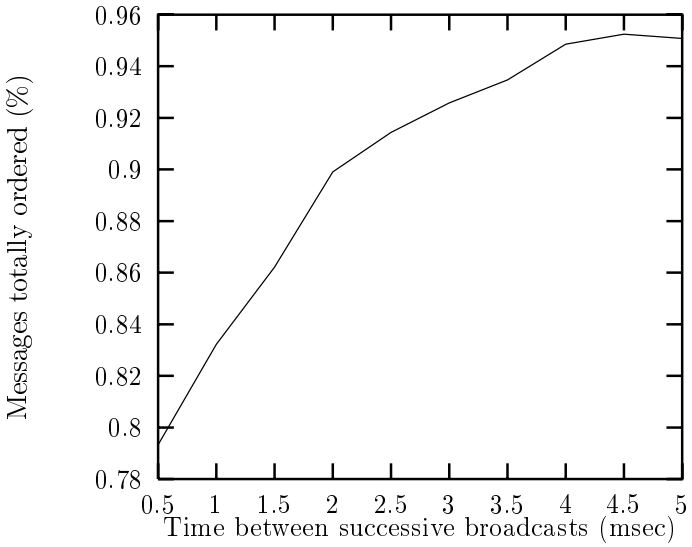

Figure 5.1: Spontaneous total order property

The approaches presented next assume that in order to deliver a message, an Atomic Broadcast algorithm proceeds in two phases. In the first phase, the message is propagated to all processes, and in the second phase, processes determine the order in which messages have to be delivered.[1]

In addition to the *propagation phase* and the *order phase*, we also consider a *check phase*, and a *treatment phase* to characterise and compare optimistic broadcast approaches. The check phase determines whether the spontaneous total order property holds, and the treatment phase represents the processing executed by the application upon A-delivering a message (e.g., the treatment done by a replica in the context of

[1]Indeed, several Atomic Broadcast algorithms based on non-centralised control are structured in a propagation and order phases [AM92, CT96].

Active Replication [GS97]). The treatment phase has been included to help compare the overhead introduced by some of the optimistic techniques.

To keep the presentation simple, we focus our attention on one message ordered at a time. However, the optimistic approaches presented can also be used when messages are ordered in batches. The beginning of the propagation phase is determined by the A-broadcast of a message, and the treatment phase starts on a process when the message is A-delivered by this process.

To assess the efficiency of our optimistic approaches, we associate with each approach a "cost" Λ, which includes ordering and processing costs.

5.1.1 Classical Atomic Broadcast Algorithm with Conservative Treatment

This approach serves as a reference for the optimistic techniques presented next. Consider that m is a message A-broadcast by a process p. Process p first sends m to all processes (including itself), and once m is received by some processes, a protocol is used to decide on the delivery order of m. The number of processes that have to receive m so that order can be decided depends on the protocol. A process q only A-delivers message m after m's order is known by q.

Figure 5.2 depicts the propagation, order, and treatment phases involved in the Classical Algorithm with Conservative Treatment approach. In this case, the cost is $\Lambda_{CC} = \Lambda_p + \Lambda_o + \Lambda_t$, where Λ_p represents the cost of the propagation phase, Λ_o the cost of the ordering phase, and Λ_t the cost of the treatment phase.

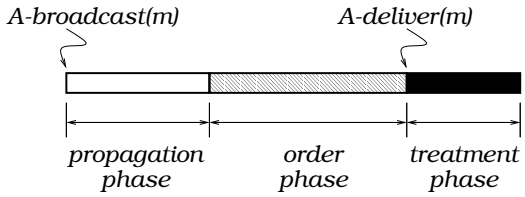

Figure 5.2: Classical approach

5.1.2 Optimistic Atomic Broadcast Algorithm with Conservative Treatment

The Optimistic Algorithm with Conservative Treatment approach assumes that checking whether the spontaneous total order property holds or not is cheaper than actually totally ordering messages. Thus, instead of executing the order phase after receiving a message m, processes try to determine whether m is received in the same order by all receivers. If this is the case, m can be delivered. Otherwise, processes have to agree on the order m should be delivered. Figure 5.3 depicts the Optimistic Algorithm with Conservative Treatment approach, with the check phase, and α, the probability that the spontaneous total order property holds.

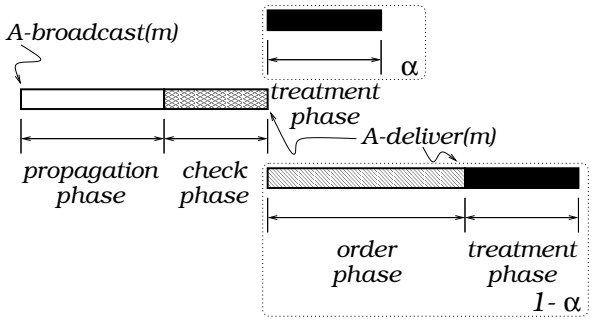

Figure 5.3: Optimistic Algorithm approach

The messages necessary to execute the check phase introduces an additional cost, and so, when the spontaneous total order property does not hold, the Optimistic Algorithm approach is more expensive than the Classical Algorithm approach (see Section 5.1.1). In this case we define the cost Λ_{OC} as $\Lambda_p + \Lambda_c + (1-\alpha)\,\Lambda_o + \Lambda_t$, where Λ_c represents the cost of the check phase. Section 5.2 presents in detail an algorithm that uses the Optimistic Algorithm approach.

An interesting optimisation would be to overlap the check with the order phases to reduce (or completely eliminate) the overhead with the delivery of messages when the spontaneous total order property does not hold.

5.1.3 Classical Atomic Broadcast Algorithm with Optimistic Treatment

A more aggressive way of exploiting the spontaneous total order property than the previous technique is as follows [KPAS99]. When a process p receives a message m, p A-delivers m to the application before executing the order phase. This way, a message is A-delivered first in a *tentative* order. Processes also execute the order phase, and once the *definitive* order for a message is known, the message is A-delivered again. Although the order phase is always performed, its execution is overlapped with the treatment phase. If the definitive order does not correspond to the tentative order, the application has to "undo" some operations and "redo" them in the correct order (see Figure 5.4).

For the Classical Atomic Broadcast Algorithm with Optimistic Treatment approach, the cost is $\Lambda_{CO} = \Lambda_p + (1-\alpha)(\Lambda_o + \Lambda_u) + \Lambda_t$, where Λ_u represents the cost for the undo phase, and either (1) $\Lambda_t \leq \Lambda_o$, or (2) if the tentative order is not the same definitive order, once the definitive order is known for some message m, the application treatment of m can be interrupted. Since messages can be received the first time in a wrong order, this approach requires the application to be able to undo the operations of the treatment phase.

As pointed out previously, this technique does not correspond to the Atomic Broadcast specification presented in Chapter 2. The new specification is defined by the primitives *Opt-broadcast*(m), *Opt-deliver*(m), and *TO-deliver*(m), which satisfy the

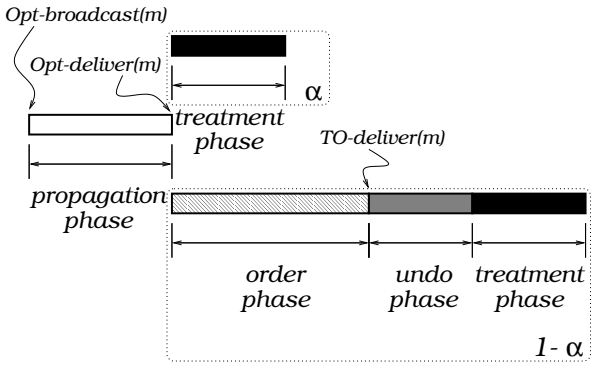

Figure 5.4: Optimistic Problem approach

following properties [KPAS99].

(Validity) If a correct process Opt-broadcasts a message m, then it eventually Opt-delivers m and TO-delivers m.

(Agreement) If a correct process Opt-delivers a message m then every correct process eventually Opt-delivers m. If a correct process TO-delivers m then every correct process eventually TO-delivers m.

(Uniform integrity) For every message m, every process Opt-delivers and TO-delivers m at most once, and only if m was previously Opt-broadcast by $sender(m)$.

(Global Order) If two correct processes p and q TO-deliver two messages m and m', then p TO-delivers m before m' if and only if q TO-delivers m before m'.

(Local Order) A process does not TO-deliver a message m before Opt-delivering m.

These properties state that every message Opt-broadcast by a correct process is eventually Opt-delivered and TO-delivered by every correct process in the system. Order is guaranteed in such a way that no process TO-delivers a message before Opt-delivering it, and every message is TO-delivered (but not necessarily Opt-delivered) in the same order by all the correct processes.

5.1.4 Optimistic Atomic Broadcast Algorithm with Optimistic Treatment

An algorithm can be devised by combining the two approaches presented before. That is, processes Opt-deliver messages as soon as they receive them, but only execute the order phase if the spontaneous total order property does not hold (see Figure 5.5).

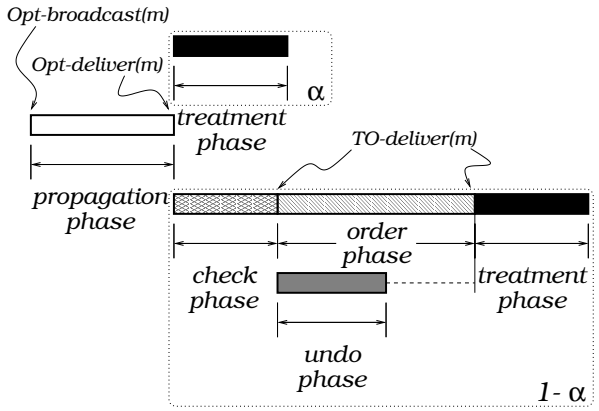

Figure 5.5: Hybrid approach

This approach has the advantages of both the Optimistic Atomic Broadcast Algorithm with Conservative Treatment and the Classical Atomic Broadcast Algorithm with Optimistic Treatment: it overlaps the treatment phase with the execution of the check phase, and it takes advantage of system properties to deliver messages (in the definitive order) fast. In this case, the cost is $\Lambda_{OO} = \Lambda_p + (1-\alpha)(\Lambda_c + \max(\Lambda_o, \Lambda_u)) + \Lambda_t$, where (1) $\Lambda_t \leq \Lambda_c$, or (2) if the tentative order of m is not correct, the application treatment started by the Opt-delivery of some message m can be interrupted.

Table 5.1 shows the approaches presented in this section. Each approach is characterised by its cost.

Approach	**Cost**
Conservative Treatment:	
Classical Atomic Broadcast Alg.	$\Lambda_{CC} = \Lambda_p + \Lambda_o + \Lambda_t$
Optimistic Atomic Broadcast Alg.	$\Lambda_{OC} = \Lambda_p + \Lambda_c + (1-\alpha)\,\Lambda_o + \Lambda_t$
Optimistic Treatment:	
Classical Atomic Broadcast Alg.	$\Lambda_{CO} = \Lambda_p + (1-\alpha)(\Lambda_o + \Lambda_u) + \Lambda_t$
Optimistic Atomic Broadcast Alg.	$\Lambda_{OO} = \Lambda_p + (1-\alpha)(\Lambda_c + \max(\Lambda_o, \Lambda_u)) + \Lambda_t$

Table 5.1: Cost of the various approaches

5.1.5 A Strawman Analysis of the Degrees of Optimism

From the cost characterisation presented in the previous sections, and by making some simplifying assumptions, we can evaluate and draw some conclusions about the behaviour of each optimistic approach.

In the following, we "quantify" the cost of a phase by its latency (also known as communication steps). The values taken for the propagation, order, and check phases

are based on best case analysis (i.e., no failures nor process suspicions). The cost for a process to propagate a message to all the other processes (broadcast), supposing n processes, is $\Lambda_p = 1$. To check whether the spontaneous total order property holds, we consider that one process is chosen *a priori* as the coordinator and the other processes send to the coordinator a list with the order of the messages received. The coordinator determines whether the spontaneous total order property holds and informs all processes. Therefore, $\Lambda_c = 2$. The order phase can be implemented using the *optimised* Chandra and Toueg Consensus algorithm with unreliable failure detectors of class $\diamond S$ [CT96].[2] In the best case we have $\Lambda_o = 3$.

We proceed considering two cases: (a) $\Lambda_t = \Lambda_u = 0$ (i.e., the cost for the treatment phase and the undo phase are equal), and (b) $\Lambda_t = \Lambda_u = \Lambda_o$ (i.e., the treatment, undo, and ordering phases have the same cost). In case (a), the costs of the treatment and the undo phases are very low (e.g., local resources are much faster than the network), and in case (b), the costs of the treatment and undo phases are high, relatively to the send to all, check, and order phases.

Simple calculations lead to the relations shown in Table 5.2, which are depicted in Figures 5.6 and 5.7.

	$\Lambda_t = \Lambda_u = 0$	$\Lambda_t = \Lambda_u = \Lambda_o$
$\Lambda_{CC}/\Lambda_{OC}$	$4/(6-3\alpha)$	$7/(9-3\alpha)$
$\Lambda_{CC}/\Lambda_{CO}$	$4/(4-3\alpha)$	$7/(10-6\alpha)$
$\Lambda_{CC}/\Lambda_{OO}$	$4/(6-5\alpha)$	$7/(9-5\alpha)$

Table 5.2: Relationships between degrees of optimism

Figures 5.6 and 5.7 show that the Classical Algorithm with Optimistic Treatment and the Optimistic Algorithm with Optimistic Treatment approaches "perform better" (in terms of latency) than the Optimistic Algorithm with Conservative Treatment approach. This is in part because even when the spontaneous total order property holds, a message can only be delivered using the Optimistic Algorithm with Conservative Treatment approach after the check phase has terminated, which is not the case with the other techniques. However, this analysis does not take into account resources utilisation (i.e., processor and network). If resources were considered, the results might have been different. The reason is that the Optimistic Algorithm with Conservative Treatment approach never has to undo operations (i.e., it generates less processor activity), and only orders messages when the spontaneous total order property does not hold (i.e., it generates less network activity). Furthermore, as already stated, there is a fundamental difference between the Optimistic Algorithm with Conservative Treatment approach and the approaches based on optimistic treatment, in that the latter can only be used when the application is able to undo operations.

From Figures 5.6 and 5.7, the Classical Algorithm with Optimistic Treatment approach is more efficient than the Optimistic Algorithm with Optimistic Treatment

[2] The *optimised* Chandra and Toueg Consensus algorithm consists in eliminating the first phase of the algorithm, when processes send their initial values to the coordinator (see the Appendix), and having the coordinator propose its initial value as estimate [Sch97].

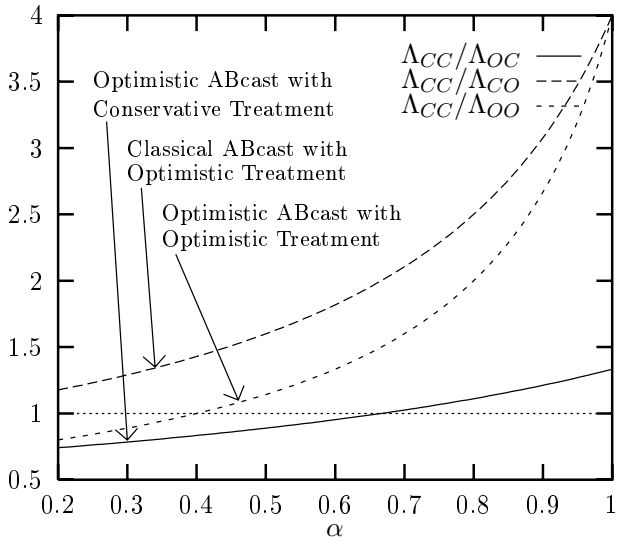

Figure 5.6: Degrees of optimism ($\Lambda_t = \Lambda_u = 0$)

approach when the cost of the treatment and undo phases are low, and worse when they are high. This is explained by the fact that the Optimistic Algorithm with Optimistic Treatment approach allows an overlap between the order and the undo phases, with the additional cost of a check phase. The Classical Algorithm with Optimistic Treatment approach does not have the additional check phase cost, but treats the order and the undo phases sequentially. If the cost of the undo phase is zero, the Optimistic Algorithm with Optimistic Treatment approach does not have any advantage over the Classical Algorithm with Optimistic Treatment approach, and, actually, further augments the overall cost to deliver a message. Finally, for values of α very close to one, both approaches based on optimistic treatment have a similar behaviour.

5.2 Optimistic Atomic Broadcast Algorithm

In this section, we present in detail an algorithm that exploits the Optimistic Approach introduced in the previous section: the Optimistic Atomic Broadcast (OPT-ABcast) algorithm. Our interest in the Optimistic Approach comes from the fact that the OPT-ABcast algorithm can replace classical implementations of Atomic Broadcast in the Database State Machine without further modifications in the way transactions are processed (see [KPAS99] for a database replication protocol based on optimistic treatment).

This section considers an asynchronous system model. Processes communicate by message passing through FIFO Quasi-Reliable channels, and have the crash-stop mode of failure. The system is augmented with failure detectors to allow us to solve Consensus (Chapter 2).

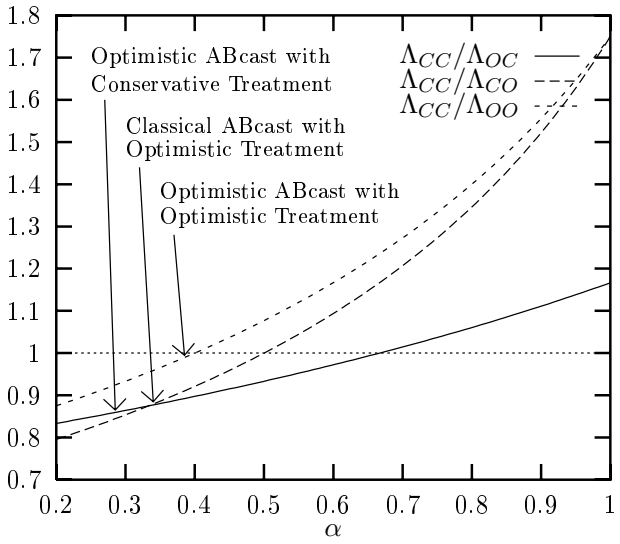

Figure 5.7: Degrees of optimism ($\Lambda_t = \Lambda_u = \Lambda_o$)

5.2.1 Overview of the Algorithm

In the OPT-ABcast algorithm, processes progress in a sequence of stages. Messages can be delivered "during" a stage or at the "end" of a stage, and the key idea is that during a stage, messages can be delivered faster than at the end of a stage. Figure 5.8 depicts the OPT-ABcast algorithm when messages are delivered during a stage k.[3] In order for a process p to deliver messages during stage k, p has to determine whether the spontaneous total order property holds. Process p determines whether this property holds by exchanging information about the order in which messages are received. Once p receives this order information from all the other processes, p uses a *prefix* function to determine whether there is a non-empty common sequence of messages received by all processes.

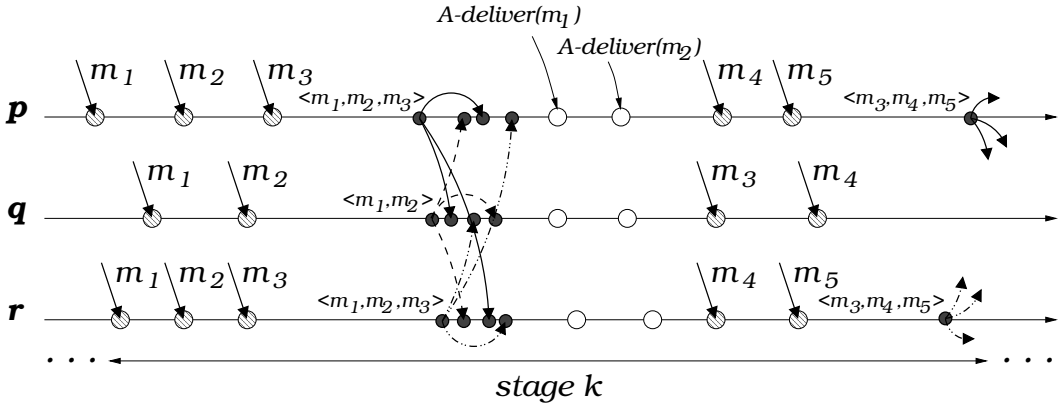

Figure 5.8: Overview of the OPT-ABcast algorithm (stage k)

[3] In Figures 5.8 and 5.9, $\langle m_1, m_2, ... \rangle$ denotes the sequence $m_1, m_2, ...$ of messages.

Figure 5.9 depicts the way the OPT-ABcast algorithm proceeds from stage k to stage $k+1$. Whenever the spontaneous total order property does not hold, processes terminate the current stage, and start a new one. The termination of a stage involves the execution of a Consensus, which can lead to the delivery of messages. Process failures are discussed in Section 5.3.3.

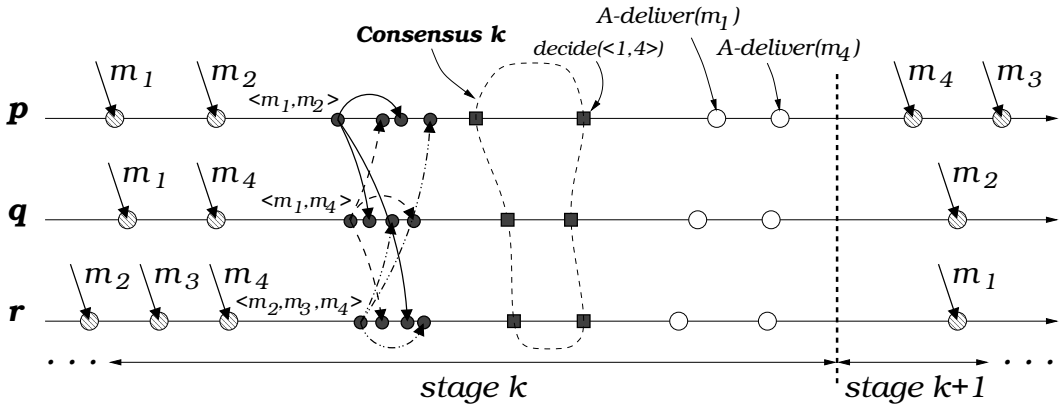

Figure 5.9: Overview of the OPT-ABcast algorithm (stages k and $k+1$)

The notion of efficiency is captured by the delivery latency defined in Section 4.3.2, which informally defines a measure of the synchronisation needed by the OPT-ABcast algorithm to deliver messages. We show that messages delivered during a stage have a deliver latency equal to 2, and messages delivered at the end of a stage have a deliver latency equal to 4. The additional cost *payed* to deliver messages at the end of a stage comes from the Consensus execution.

Known Atomic Broadcast implementations for the asynchronous model augmented with failure detectors deliver messages with a deliver latency equal to 3 [CT96]. This means that if the spontaneous total order property is violated too frequently, the OPT-ABcast algorithm may become inefficient. However, in case the spontaneous total order property holds frequently, messages can be delivered efficiently using the OPT-ABcast algorithm.

5.2.2 Additional Notation

The Optimistic Atomic Broadcast algorithm presented in the next section handles sequences of messages. In the following we define some terminology needed for the presentation of the algorithm.

A sequence of messages is denoted by $seq = \langle m_1, m_2, \dots \rangle$. We define the operators $\oplus$ and $\ominus$ for concatenation and decomposition of sequences. Let seq_i and seq_j be two sequences of messages. Then, $seq_i \oplus seq_j$ is the sequence of all the messages in seq_i followed by the sequence of all the messages in seq_j, and $seq_i \ominus seq_j$ is the sequence of all the messages in seq_i that are not in seq_j. So, the sequence $seq_i \ominus seq_j$ does not contain any message in seq_j. The prefix function $\odot$ applied to a set of sequences

returns the longest common sequence that is a prefix of all the sequences, or the empty sequence denoted by ϵ.

For example, if $seq_i = \langle m_1, m_2, m_3 \rangle$ and $seq_j = \langle m_1, m_2, m_4 \rangle$, then $seq_i \oplus seq_j = \langle m_1, m_2, m_3, m_1, m_2, m_4 \rangle$, $seq_i \ominus seq_j = \langle m_3 \rangle$, and $\odot(seq_i, seq_j) = \langle m_1, m_2 \rangle$.

5.2.3 Detailed Algorithm

Algorithm 2 (page 89) solves Atomic Broadcast. Processes executing Algorithm 2 progress in a sequence of local stages numbered $1, ..., k, ...$ Messages A-delivered by a process *during stage k* are included in the sequence $stgA\_deliver^k$. These messages are A-delivered without the cost of Consensus. Messages A-delivered by a process *at the end of stage k* are included in the sequence $endA\_deliver^k$. These messages are A-delivered with the cost of a Consensus execution. We say that a message m is A-delivered *in stage k* if m is A-delivered either during stage k or at the end of stage k.

Every stage k is terminated by a Consensus to decide on a sequence of messages, denoted by $msgStg^k$. Algorithm 2 guarantees that if a correct process starts Consensus (by invoking the *propose* primitive), all correct processes also start Consensus. Notice that if not all correct processes invoke the *propose* primitive in the k-th Consensus execution, then Consensus termination cannot be ensured.

The sequence $msgStg^k$ contains all message that are A-delivered in stage k (i.e., during stage k and at the end of stage k) by every process that reaches the end of stage k. Process p starts stage $k+1$ once it has A-delivered all messages in $endA\_deliver^k$, where $endA\_deliver^k = msgStg^k \ominus stgA\_deliver^k$.

The correctness of Algorithm 2 is based on two properties:

1. for any correct processes p and q, all the messages A-delivered by p in stage k are also A-delivered by q in stage k (i.e., $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$), and

2. every sequence of messages A-delivered by some process p in stage k before p executes Consensus k is a non-empty prefix of the sequence decided in Consensus k (i.e., $stgA\_deliver_p^k$ is a prefix of $msgStg^k$).

All tasks in Algorithm 2 execute concurrently. At each process p, tasks *GatherMsgs* (lines 11-12) and *TerminateStage* (lines 25-35) are started at initialisation time. Task $StgDeliver^k$ (lines 13-24) is started by p when p begins stage k. Lines 20 and 21 in task $StgDeliver^k$ are atomic, that is, task $StgDeliver^k$ is not interrupted (by task *TerminateStage*) after it has executed line 20 and before having executed line 21. Process p in stage k manages the following sequences.

- $R\_delivered_p$: contains all messages R-delivered by p up to the current time,
- $A\_delivered_p$: contains all messages A-delivered by p up to the current time,

- $stgA\_deliver_p^k$: is the sequence of messages A-delivered by p during stage k, up to the current time,
- $endA\_deliver_p^k$: is the sequence of messages A-delivered by p at the end of stage k.

When p wants to A-broadcast message m, p executes *R-broadcast*(m) (line 9). After p R-delivers a message m (line 11), p includes m in $R\_delivered_p$, and eventually executes task $StgDeliver^k$ (line 13). At task $StgDeliver^k$, p sends a sequence of messages that it has not A-delivered yet to all processes (line 14), and waits for such sequence from all processes (line 15). The next actions executed by p depend on the messages it receives at the *wait* statement (line 15).

1. If p receives a sequence from all processes, and there is a non-empty prefix common to all these sequences, then p A-delivers the messages in the common prefix (line 20). If not, p R-broadcasts message (k, ENDSTG) to terminate the current stage k (line 23).
2. Once p R-delivers message (k, ENDSTG) at line 25, p terminates task $StgDeliver^k$ (line 26), and starts the k-th Consensus execution (line 27), proposing a sequence of all messages p has R-delivered up to the current time but not A-delivered in any stage k', $k' < k$. Upon deciding for Consensus k (line 28), p builds the sequence $endA\_deliver^k$ (line 29) and A-delivers the messages in $endA\_deliver^k$ (line 30). Process p then starts stage $k+1$ (lines 32-35).

5.2.4 Proof of Correctness

The correctness of the OPT-ABcast algorithm follows from Propositions 5.1 (Agreement), 5.2 (Total Order), 5.3 (Validity), and 5.4 (Integrity). In the following proofs, we consider the number of times that processes execute lines 13-21 in a given stage. Hereafter, $stgA\_deliver_p^{k,l_k}$ denotes the value of $stgA\_deliver_p^k$ after p executed line 21 for the l_k-th time in stage k, $l_k > 0$, and $stgA\_deliver_p^{k,0}$ denotes ϵ (the value of $stgA\_deliver_p^k$ before p executes lines 13-21 for the first time). Likewise, $prefix_p^{l_k}$, respectively $msgSeq^{l_k}$, denotes the value of $prefix_p$, respectively $msgSeq$, after p executed line 17, respectively 15, for the l_k-th time in a given stage. In the proofs presented next "$\forall p$" means "$\forall p \in \Pi$".

The proofs of Lemmata 5.1 and 5.3 use the FIFO property of the communication channels to conclude that for any process p that executes the l-th iteration of line 17, $prefix_p^l = \odot_{\forall q} msgSeq_q^l$. This statement holds since when p executes the l-th iteration of line 17, p has received l messages of the type $(-, msgSeq_q)$ from every process q. The FIFO channels guarantee that all processes that execute the l-th iteration of line 17 receive the messages $(-, msgSeq_q)$ in the same order $(-, msgSeq_q^1), (-, msgSeq_q^2), \ldots, (-, msgSeq_q^l)$ from every q.

Lemma 5.1 *If p and q are two processes that execute the l_k-th iteration of line 21 in stage k, then $stgA\_deliver_p^{k,l_k} = stgA\_deliver_q^{k,l_k}$.*

Algorithm 2 OPT-ABcast algorithm

1: Initialisation (see Section 5.2.3 for a description of the variables):
2: $R\_delivered \leftarrow \epsilon$
3: $A\_delivered \leftarrow \epsilon$
4: $k \leftarrow 1$
5: $stgA\_deliver^k \leftarrow \epsilon$
6: $endA\_deliver^k \leftarrow \epsilon$
7: **fork tasks** { *GatherMsgs*, $StgDeliver^1$, *TerminateStage* }

8: To execute *A-broadcast*(m):

9: *R-broadcast*(m)

10: *A-deliver*$(-)$ occurs as follows:

11: **when** *R-deliver*(m) *{**Task** GatherMsgs}*
12: $R\_delivered \leftarrow R\_delivered \oplus \langle m \rangle$

13: **when** $(R\_delivered \ominus A\_delivered) \ominus stgA\_deliver^k \neq \epsilon$ *{**Task** $StgDeliver^k$}*
14: send $(k, (R\_delivered \ominus A\_delivered) \ominus stgA\_deliver^k)$ to all
15: **wait until** for $[\forall q \in \Pi :$ received $(\underline{k}, msgSeq_q)$ from q **or** $\mathcal{D}_p \neq \emptyset]$
16: $\pi = \{ q \mid p$ received $(k, msgSeq_q)$ from $q \}$
17: $prefix \leftarrow \odot_{\forall q \in \pi} msgSeq_q$
18: **if** $\pi = \Pi$ **and** $prefix \neq \epsilon$ **then**
19: $stgDeliver \leftarrow prefix \ominus stgA\_deliver^k$
20: [deliver all messages in *stgDeliver* according to their order in *stgDeliver*;
21: $stgA\_deliver^k \leftarrow stgA\_deliver^k \oplus prefix$]
22: **else**
23: *R-broadcast*(k, EndStg)
24: **end task**

25: **when** *R-deliver*$(\underline{k}, \underline{\text{EndStg}})$ *{**Task** TerminateStage}*
26: **terminate task** $StgDeliver^k$, if executing
27: *propose*$(k, R\_delivered \ominus A\_delivered)$
28: **wait until** *decide*$(\underline{k}, msgStg^k)$
29: $endA\_deliver^k \leftarrow msgStg^k \ominus stgA\_deliver^k$
30: deliver all messages in $endA\_deliver^k$ following their order in $endA\_deliver^k$
31: $A\_delivered \leftarrow A\_delivered \oplus (stgA\_deliver^k \oplus endA\_deliver^k)$
32: $k \leftarrow k + 1$
33: $stgA\_deliver^k \leftarrow \epsilon$
34: $endA\_deliver^k \leftarrow \epsilon$
35: **fork task** $StgDeliver^k$

PROOF. We first show that for any l, $0 < l \leq l_k$, $prefix_p^l = prefix_q^l$. Since p and q execute line 21 for the l-th time in stage k, p and q receive a message of the type $(k, msgSeq)$ from every process in the l-th iteration of lines 15. From line 17 and the fact that communication between processes follows a FIFO order, $prefix_p^l = \odot_{\forall r} msgSeq_r^l$, and $prefix_q^l = \odot_{\forall r} msgSeq_r^l$, where $msgSeq_r^l$ is the l-th message of the type $(k, msgSeq_r)$ received from process r, and we conclude that $prefix_p^l = prefix_q^l$. From line 21, $stgA\_deliver^{k,l} = stgA\_deliver^{k,l-1} \oplus prefix^l$, and a simple induction on l_k leads to $stgA\_deliver_p^{k,l_k} = stgA\_deliver_q^{k,l_k}$. □

Lemma 5.2 *If some process p executes line 21 l times, then all processes in Π execute the send statement at line 14 l times.*

PROOF. This follows directly from the algorithm since p can only execute line 21 after receiving message $(k, msgSeq)$ (line 15) from all processes. Thus, if p executes line 21 l times, it receives message $(k, msgSeq)$ from all processes l times, and from the no creation property of Reliable Channels, all processes execute the $send(k, -)$ statement at line 14 l times. □

Lemma 5.3 *For any process p, and all $k \geq 1$, if p executes $decide(k, msgStg^k)$, then (a) $stgA\_deliver_p^k$ is a prefix of $msgStg^k$, and (b) $stgA\_deliver_p^k$ does not contain the same message more than once.*

PROOF. Assume that p executes $decide(k, msgStg^k)$. By uniform validity of Consensus, there is a process q that executed $propose(k, R\_delivered_q \ominus A\_delivered_q)$, such that $R\_delivered_q \ominus A\_delivered_q = msgStg^k$. Let l_k be the number of times that p executes line 21 before executing $decide(k, -)$. From Lemma 5.2, all processes in Π execute the *send* statement at line 14 l_k times.

We show by induction on l_k that $stgA\_deliver_p^{k,l_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$, and $stgA\_deliver_p^{k,l_k}$ does not contain the same message more than once. BASE STEP. ($l_k = 0$) In this case, $stgA\_deliver_p^{k,0} = \epsilon$ and the lemma is trivially true. INDUCTIVE STEP. Assume that the lemma holds for all l'_k, $0 < l'_k < l_k$. We show that $stgA\_deliver_p^{k,l_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$, and $stgA\_deliver_p^{k,l_k}$ does not contain the same message more than once. By line 21, $stgA\_deliver_p^{k,l_k} = stgA\_deliver_p^{k,(l_k-1)} \oplus prefix_p^{l_k}$. Since communication channels are FIFO, any message sent by some process r in the l_k-th execution of $send(k, msgSeq_r^{l_k})$ (line 14) is received by p in the l_k-th execution of the statement $receive(k, msgSeq_r^{l_k})$ (line 15), and therefore, after p executes line 17, $prefix_p^{l_k} = \odot_{\forall r} msgSeq_r^{l_k}$. From lines 14 and 15, $msgSeq_r^{l_k} = (R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,(l_k-1)}$, and so, $prefix_p^{l_k} = \odot_{\forall r}((R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,(l_k-1)})$. By Lemma 5.1, we have $prefix_p^{l_k} = \odot_{\forall r}((R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_p^{k,(l_k-1)})$. So, $stgA\_deliver_p^{k,l_k} = stgA\_deliver_p^{k,(l_k-1)} \oplus (\odot_{\forall r}(R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_p^{k,(l_k-1)})$. From the induction hypothesis, item (a), we have that $stgA\_deliver_p^{k,(l_k-1)}$ is a prefix

of $R\_delivered_q \ominus A\_delivered_q$. Furthermore, from item (b) of the induction hypothesis, all messages in $stgA\_deliver_p^{k,(l_k-1)}$ are unique. Thus, $stgA\_deliver_p^{k,l_k} = \odot_{\forall r}(R\_delivered_r \ominus A\_delivered_r)$,[4] and therefore, $stgA\_deliver_p^{k,l_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$. It also follows that $stgA\_deliver_p^{k,l_k}$ does not contain the same message more than once. For a contradiction, assume that message m is more than once in $stgA\_deliver_p^{k,l_k}$. Thus, for every process r, m is more than once in $R\_delivered_r$. From the algorithm, lines 11 and 12, m has been R-delivered more than once by r, contradicting uniform integrity of Reliable Broadcast. □

Lemma 5.4 *For any two correct processes p and q, and all $k \geq 1$, if p executes line 30 in stage k, then q executes line 30 in stage k.*

Proof. If p executes line 30 in stage k, then p executes the $decide(k, msgStg^k)$ statement at line 28, and the $propose(k, -)$ statement at line 27. Therefore, p R-delivers a message of the type (k, EndStg) at line 25. By the agreement property of Reliable Broadcast, q eventually R-delivers message (k, EndStg), and executes the $propose(k, -)$ statement at line 27. By agreement of Consensus, q executes the $decide(k, msgStg^k)$ statement, and line 30. □

Lemma 5.5 *For any two processes p and q, and all $k \geq 1$, if both p and q execute line 29, then $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$.*

Proof. From line 29, $endA\_deliver_p^k = msgStg^k \ominus stgA\_deliver_p^k$, and therefore, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_p^k \oplus (msgStg^k \ominus stgA\_deliver_p^k)$. By Lemma 5.3, $stgA\_deliver_p^k$ is a prefix of $msgStg^k$, and so, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = msgStg^k$. From a similar argument, we have $stgA\_deliver_q^k \oplus endA\_deliver_q^k = msgStg^k$. We conclude that $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$. □

Lemma 5.6 *For any process p, and all $k \geq 1$, if message $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$ then there is no k', $k' < k$, such that $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$.*

Proof. The proof is by contradiction. Assume that there exist a process p, a message m, some k, and some $k' < k$, such that $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$. We distinguish two cases: (a) $m \in stgA\_deliver_p^k$, or (b) $m \in endA\_deliver_p^k$. Note that from line 29, it cannot be that $m \in stgA\_deliver_p^k$ and $m \in endA\_deliver_p^k$.

Case (a). From lines 21, 17 and 15 $stgA\_deliver_p^k$ is a common non-empty prefix among the messages of the type $(k, msgSeq)$ received by p from all processes.

[4] Let seq_i and seq_j be two sequences such that seq_i is a prefix of seq_j, and messages in seq_j are unique. It can be shown that $seq_i \oplus (seq_j \ominus seq_i) = seq_j$.

Thus p has received the message $(k, msgSeq_p)$ (i.e., a message that p sent to itself), such that $m \in msgSeq_p$. But $msgSeq_p = R\_delivered_p \ominus A\_delivered_p$ (line 14), and so, $m \notin A\_delivered_p$. When p executes line 14 at stage k, $A\_delivered_p = \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$. This follows from line 31, the only line where $A\_delivered$ is updated. Therefore, $m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$, contradicting the fact that there is a $k' < k$ such that $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$.

Case (b). From line 29, $m \in msgStg^k$, and from line 28, and validity of Consensus, there is a process q that executes $propose(k, R\_delivered_q \ominus A\_delivered_q)$ such that $m \in R\_delivered_q \ominus A\_delivered_q$. So, $m \notin A\_delivered_q$. Since when q executes line 27, $A\_delivered_q = \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i), m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i)$, and from Lemma 5.5 $\oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i) = \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i)$. Thus, we conclude that $m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$, a contradiction that concludes the proof. □

Proposition 5.1 (AGREEMENT). *If a correct process p A-delivers a message m, then every correct process q eventually A-delivers m.*

PROOF: Consider that p has A-delivered message m in stage k. We show that q also A-delivers m in stage k. There are two cases to consider: (a) p A-delivers messages in $endA\_deliver_p^k$, and (b) p does not A-deliver messages in $endA\_deliver_p^k$.

Case (a). From Lemma 5.4 and the fact that p A-delivers messages in $endA\_deliver_p^k$, q A-delivers messages in $endA\_deliver_q^k$, and from Lemma 5.5, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$. Since p A-delivers m in stage k, $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and so, $m \in stgA\_deliver_q^k \oplus endA\_deliver_q^k$. Therefore, q either A-delivers m at line 20 (in which case $m \in stgA\_deliver_q^k$), or at line 30 (in which case $m \in stgA\_deliver_q^k$).

Case (b). Since p does not A-deliver messages in $endA\_deliver_p^k$, from Lemma 5.4, no correct process q A-delivers messages in $endA\_deliver_q^k$. However, m is A-delivered in stage k by p, and so, it must be that $m \in stgA\_deliver_p^k$. Assume that $m \in stgA\_deliver_p^{k,l_k}$, where l_k is such that for any $l'_k < l_k$, $m \notin stgA\_deliver_p^{k,l'_k}$. Therefore, p executes the l_k-th iteration of line 21 in stage k, and we claim that q also executes the l_k-th iteration of line 21 in stage k. The claim is proved by contradiction. From the algorithm, q executes $R\text{-}broadcast(k, -)$. By agreement and validity of Reliable Broadcast, every correct process R-delivers the message (k, ENDSTG) and executes $propose(k, -)$. By agreement and termination of Consensus, every correct process decides on Consensus k, and eventually A-delivers messages in $endA\_deliver^k$, contradicting the fact that no correct process A-delivers messages in $endA\_deliver^k$, and concluding the proof of the claim. Since p and q execute the l_k-th iteration of line 21 in stage k, and $m \in stgA\_deliver_p^{k,l'_k}$, from Lemma 5.1,

$m \in stgA\_deliver_q^{k,l'_k}$, and from lines 20-21, q A-delivers m. □

Proposition 5.2 (Total Order). *If correct processes p and q both A-deliver messages m and m', then p A-delivers m before m' if and only if q A-delivers m before m'.*

Proof: Assume that p A-delivers message m in stage k, and m' in stage k', $k' > k$. Therefore, $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and $m' \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$, and it follows immediately from Lemma 5.5 that q A-delivers m before m'. Now, assume that m and m' are A-delivered by p in stage k. Thus, m precedes m' in $stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and by Lemma 5.5, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$.

We claim that if m precedes m' in $stgA\_deliver_q^k \oplus endA\_deliver_q^k$, then q A-delivers m before m'. If $m, m' \in stgA\_deliver_q^k$ (respectively $m, m' \in endA\_deliver_q^k$), then, from task $stgDeliver^k$, line 20 (respectively *TerminateStage*, line 30), q A-delivers m before m'. Thus, consider that $m \in stgA\_deliver_q^k$ and $m' \in endA\_deliver_q^k$. To reach a contradiction, assume that q A-delivers m' before m. Before A-delivering m at line 20, q executes line 26 and terminates task $stgA\_deliver_q^k$, and so, m cannot be A-delivered in stage k, contradicting that m and m' are A-delivered in stage k, and concluding the proof of the lemma. □

Lemma 5.7 *If a correct process p executes line 25 in stage k, then every correct process q executes line 25 in stage k.*

Proof. The proof is by induction on k. Base step. ($k = 1$) Initially, all correct processes are in stage 1. Thus, if p executes line 25 in stage 1 and R-delivers message $(1, \text{EndStg})$, by the agreement property of Reliable Channels, every correct process eventually executes line 25 and R-delivers message $(1, \text{EndStg})$. Inductive step. Assume that if a correct process p executes line 25 in stage $k-1$, then every correct process q executes line 25 in stage $k-1$. We show that if p executes line 25 in stage k, then q also executes line 25 in stage k. From the algorithm and the termination property of Consensus, after R-delivering message $(k-1, \text{EndStg})$, all correct processes eventually terminate Consensus in stage $k-1$ and execute lines 32-35, starting stage k. Since p R-delivers message (k, EndStg), by agreement of Reliable Channels, every correct process q R-delivers message (k, EndStg). □

Lemma 5.8 *No correct process p has a task $stgDeliver_p^k$, $k > 0$, that is permanently blocked in the wait statement of line 15.*

Proof. For a contradiction, consider that there exists a correct process p such that for some $l_k > 0$, task $stgDeliver_p^k$ is permanently blocked at the l_k-th iteration of line 15. Therefore, (a) there is a process q such that p never receives the message $(k, msgSeq)$ for the l_k-th time from q and (b) $q \notin \mathcal{D}_p$. From (b), and the completeness property of $\mathcal{D}_p$, q is a correct process. From Lemma 5.7, if p executes line 25 in stage

k, then q executes line 25 in stage k, but since p never receives $(k, msgSeq)$ for the l_k-th time from q, by the no loss property of Reliable Channels, q does not send message $(k, msgSeq)$ for the l_k-th time to p (line 14).

We now prove the following claim: if q does not execute $send(k, msgSeq)$ for the l_k-th time, q executes $R\text{-}deliver(k, \text{ENDSTG})$. When p executes the wait statement for the l_k-th time in stage k, there exists a message m such that $m \in (R\_delivered_p \ominus A\_delivered_p) \ominus stgA\_deliver_p^{k,(l_k-1)}$. So, (1a) $m \notin stgA\_deliver_p^{k,(l_k-1)}$, and from line 31, (1b) $m \notin \oplus_{i=1}^{k-1} stgA\_deliver_p^i \oplus endA\_deliver_p^i$. If q does not send the message $(k, msgSeq)$ for the l_k-th time to p, then either (i) $(R\_delivered_q \ominus (\oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i))) \ominus stgA\_deliver_q^{k,(l_k-1)}$ is empty (line 13) or (ii) task $stgDeliver_q^k$ is terminated before q sends message $(k, msgSeq)$ for the l_k-th time to p (i.e., q terminates stage k). Furthermore, since p executes line 15 for the l_k-th time, p has executed the $(l_k - 1)$-th iteration of lines 13-21, and received a message from all processes at line 15 for the $(l_k - 1)$-th time. Thus, every process executes the send statement at line 14 at least $l_k - 1$ times, and, from Lemma 5.1, (2a) $stgA\_deliver_p^{k,(l_k-1)} = stgA\_deliver_q^{k,(l_k-1)}$. From Lemma 5.5, (2b) for all $k', 1 \leq k' < k$, $stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'} = stgA\_deliver_q^{k'} \oplus endA\_deliver_q^{k'}$. From (1a) and (2a), we conclude that $m \notin stgA\_deliver_q^{k,(l_k-1)}$, and, from (1b) and (2b), $m \notin \oplus_{i=1}^{k-1} stgA\_deliver_q^i \oplus endA\_deliver_q^i$. Since q does not send message $(k, msgSeq)$ for the l_k-th time to p at line 14, m will never be in $R\_delivered_q$. However, by the agreement property of Reliable Broadcast, eventually $m \in R\_delivered_q$ (item (i) of the claim is false), and so, task $stgDeliver_q^k$ is terminated at line 24 or 26 before q sends message $(k, msgSeq)$ for the l_k-th time to p (item (ii) of the claim is true), and q executes $R\text{-}deliver(k, \text{ENDSTG})$, concluding our claim.

By the agreement of Reliable Broadcast, p eventually R-delivers message (k, ENDSTG), and so, p executes line 26 and terminates task $stgDeliver_p^k$, contradicting our initial hypothesis that task $stgDeliver_p^k$ remains permanently blocked. □

Proposition 5.3 (VALIDITY). *If a correct process p A-broadcasts a message m, then p eventually A-delivers m.*

PROOF: For a contradiction, assume that p A-broadcasts m but never A-delivers it. From Proposition 5.1, no correct process A-delivers m. Since p A-broadcasts m, it R-broadcasts m, and from the validity of Reliable Broadcast, p eventually R-delivers m and includes m in $R\_delivered_p$. Since no correct process A-delivers m, $m \notin A\_delivered_p$, and for all k, $m \notin stgA\_deliver^k$, $k > 0$. From the agreement of Reliable Broadcast, there is a stage k_1 such that for all $l \geq k_1$, and every correct process q, $m \in (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^l$.

Let k_2 be a stage such that for all $l \geq k_2$ every faulty process has crashed (i.e., no faulty process executes stage l), and let $k \geq max(k_1, k_2)$. Thus, no faulty process executes stage k, and for every correct process q, $m \in (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^k$ at stage k. From Lemma 5.8, for every correct p, no task $stgDeliver_p^k$ remains permanently blocked at line 15, and if task $stgDeliver_p^k$ is terminated, task

$stgDeliver_p^{(k+1)}$ is eventually started by p. Thus, all correct processes execute the *when* statement at line 13, and there are two cases two consider: (a) for all $l_k > 0$, every process executes the *then* branch of the *if* statement at line 18 (in which case there are no faulty processes in the system), and (b) for some $l_k > 0$, there is a process r that executes the *else* branch, and R-broadcasts message (k, EndStg).

Case (a). We claim that there exists an $l'_k > 0$ such that $m \in \odot_{\forall r \in \Pi} msgSeq_r^{l'_k}$. From the algorithm, for every process r, $msgSeq_r^{l_k} = (R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,l_k}$, and so, $m \in msgSeq_r^{l_k}$. Assume, for a contradiction, that for every $l'_k > 0$, $m \notin \odot_{\forall r \in \Pi} msgSeq_r^{l'_k}$. Since $m \in msgSeq_r^{l_k}$, for all r, this can only be possible if for two processes p' and p'', m precedes some message m' in $msgSeq_{p'}^{l_k}$ and m' precedes m in $msgSeq_{p''}^{l_k}$. However, in this case, eventually, $\odot_{\forall r \in \Pi} msgSeq_r = \epsilon$, and processes do not execute the *then* branch, contradicting the assumption of case (a).

Case (b). By the validity of Reliable Broadcast, r R-delivers message (k, EndStg). From Lemma 5.7, if p reaches line 25 in stage k, then q reaches line 25 in stage k, and from agreement of Reliable Broadcast, every correct process q R-delivers (k, EndStg) and executes $propose(k, R\_delivered_q \ominus A\_delivered_q)$, such that $m \in R\_delivered_q \ominus A\_delivered_q$. By agreement and termination of Consensus, every q decides on the same $msgStg^k$, and by validity of Consensus $m \in msgStg^k$. It follows that q A-delivers m, a contradiction that concludes the proof. □

Proposition 5.4 (Uniform Integrity). *For any message m, each process A-delivers m at most once, and only if m was previously A-broadcast by $sender(m)$.*

Proof: We first show that, for any message m, each process A-delivers m only if m was previously A-broadcast by $sender(m)$. There are two cases to consider. (a) A process p A-delivers m at line 20. Thus, p received a message $(k, msgSeq_q)$ from every process q, for some k, and $m \in msgSeq_q$. From line 14, $m \in R\_delivered_q$, and from line 12, p has R-delivered m. By uniform integrity of Reliable Broadcast, $sender(m)$ R-broadcasts m, and so, $sender(m)$ A-broadcasts m. (b) Process p A-delivers m at line 30. Thus, from line 29, $m \in msgSet^k$, for some k, and p executed $decide(k, msgStg^k)$. By uniform validity of Consensus, some process q executed $propose(k, R\_delivered_q \ominus A\_delivered_q)$, such that $m \in R\_delivered_q \ominus A\_delivered_q$. From an argument similar to the one presented in item (a), $sender(m)$ A-broadcasts m.

We now show that m is only A-delivered once by p. From Lemma 5.6, it is clear that if m is A-delivered in stage k (i.e., $m \in stgA\_deliver^k \oplus endA\_deliver^k$), then m is not A-delivered in some other stage k', $k' \neq k$. It remains to be shown that m is not A-delivered more than once in stage k. There are three cases to be considered: m is A-delivered at line 20 and will not be A-delivered again (a) at line 20 or (b) at line 30, and (c) m is A-delivered at line 30 and will not be A-delivered again at line 20.

Case (a). After A-delivering m at line 20, p includes m in $stgA\_deliver_p^k$, and from

line 19, p will not A-deliver m again at line 20.

Case (b). For a contradiction, assume that m is A-delivered once at line 20 and again at line 30. Thus, when p executes line 29, $m \notin stgA\_deliver_p^k$. Since m has already been A-delivered at line 20, it follows that task *StgDeliver*^{k} is terminated after p A-delivers m at line 20 and before p executes line 21. This leads to a contradiction since lines 20 and 21 are executed atomically.

Case (c). Before executing line 30, p executes line 26, and terminates task *StgDeliver*^{k}. So, once p A-delivers some message at line 30 in stage k, no message can be A-delivered at line 20 in stage k by p. □

Theorem 5.1 *Algorithm 2 solves Atomic Broadcast.*

PROOF. Immediate from Propositions 5.1, 5.2, 5.3, and 5.4. □

5.3 Evaluation of the OPT-ABcast Algorithm

Intuitively, the key idea to evaluating the OPT-ABcast algorithm is that if Consensus is not needed to deliver some message m, but necessary to deliver some other message m', then the delivery latency of m' is greater than the delivery latency of m. Before going into details about the delivery latency of messages delivered with and without the cost of a Consensus execution (see Section 5.3.2), we present a more general result about the necessity of Consensus in the OPT-ABcast algorithm.

5.3.1 On the Necessity of Consensus

Proposition 5.5 states that in a failure free and suspicion free run, Consensus is not executed in stage k if the spontaneous total order message reception property holds permanently in k.

Lemma 5.9 *For any two processes p and q, and all $k \geq 1$, if p executes line 21 for the l_k-th time in stage k, $l_k > 0$, then q executes line 21 for the $(l_k - 1)$-th time in stage k.*

PROOF. If p executes line 21 for the l_k-th time in stage k, then p executes the *wait* statement at line 15 for the l_k-th time in stage k such that p does not suspect any process and receives a message from every process (furthermore, there is a non-empty prefix between all messages received by p). From the no creation property of Reliable Channels, every process q executes the *send* statement at line 14 for the l_k-th time in stage k. For a contradiction, assume that q does not execute line 21 for the (l_k-1)-th time. Then, q executes *R-broadcast*(k, ENDSTG) (line 23) in the l'_k iteration of lines 14-24, $l'_k \leq (l_k - 1)$, and q finishes task *StgDeliver*^{k} (line 24). Therefore, q never executes the *send* statement at line 14 for the l_k-th time, a contradiction. □

Proposition 5.5 *Let R be a failure free and suspicion free run of the OPT-ABcast algorithm. If for every two processes p and q, all $k > 0$, and all $l_k > 0$, $((R\_delivered_p \ominus A\_delivered_p) \ominus stgA\_deliver_p^{k,l_k}) \odot ((R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^{k,l_k}) \neq \epsilon$, then no process executes Consensus k in R.*

Proof. Assume that there is a process p that executes Consensus k in R. From the algorithm, p R-delivers a message of the type (k, EndStg), and by uniform integrity of Reliable Broadcast, some process q executed $R\text{-}broadcast(k, \text{EndStg})$. From line 18, either (a) q suspects some process, or (b) there is an iteration $l_k \geq 0$ of lines 14-17, such that $prefix_q^{l_k+1} = \epsilon$. Case (a) contradicts the hypothesis that no process is suspected, so it must be that $prefix_q^{l_k+1} = \epsilon$.

From Lemma 5.9 and lines 17, 14 and 15, we have $prefix_q^{l_k+1} = \odot_{\forall r} msgSeq_r^{l_k+1} = \odot_{\forall r}((R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,l_k})$, and therefore, $\odot_{\forall r}((R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,l_k}) = \epsilon$. So, there must exist two processes p and q such that $((R\_delivered_p \ominus A\_delivered_p) \ominus stgA\_deliver_p^{k,l_k}) \odot ((R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^{k,l_k}) = \epsilon$, contradicting the hypothesis. □

Thus, from Proposition 5.5, in a failure free and suspicion free run, Consensus is only necessary in stage k when the spontaneous total order property does not hold in k.

5.3.2 Delivery Latency of the OPT-ABcast Algorithm

We now discuss in more detail the efficiency of the OPT-ABcast algorithm. For every process p and all stages k, there are two cases to consider: (a) messages A-delivered by p during stage k (line 20), and (b) messages A-delivered by p at the end of stage k. The main result is that for case (a), the Optimistic Atomic Broadcast algorithm can A-deliver messages with a delivery latency equal to 2, while for case (b), the delivery latency is at least equal to 4. Since known Atomic Broadcast algorithms deliver messages with a delivery latency of at least 3, these results show the tradeoff of the Optimistic Atomic Broadcast algorithm: if the spontaneous total order message reception property only holds rarely, the OPT-ABcast algorithm is not attractive, while otherwise, the OPT-ABcast algorithm leads to smaller costs compared to known Atomic Broadcast algorithms.

Propositions 5.6 and 5.7 assess the minimal cost of the Optimistic Atomic Broadcast algorithm to A-deliver a message m. Proposition 5.6 defines a lower bound on the delivery latency of Algorithm 2 for messages A-delivered without Consensus (line 20), and Proposition 5.7 states that this bound can be reached in runs where no process A-delivers m at the end a of stage.

Proposition 5.6 *There is no run R generated by Algorithm 2 where some message m is only A-delivered at line 20 (without Consensus) and $dl^R(m) < 2$.*

PROOF. Assume that m is only A-delivered during some stage $k > 0$ (i.e., without Consensus), and let p be a process that A-delivers m in R. Process p receives a message $(k, msgSeq_q)$ from every process q such that $m \in msgSet_q$. Since q executes $send(k, (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver^k)$ such that $m \in (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver^k$, q executes $R\text{-}deliver(m)$. By the way timestamps are assigned to events (see Section 4.3.2), $ts(A\text{-}deliver_p(m)) \geq ts(R\text{-}deliver_q(m)) + 1$ (1). By uniform integrity of Reliable Broadcast, there is some process r that executes $R\text{-}broadcast(m)$, which, from Algorithm 2, is the process that executes $A\text{-}broadcast(m)$. Thus, $ts(A\text{-}broadcast_r(m)) = ts(R\text{-}broadcast_r(m))$ (2).

From (1) and (2), $ts(A\text{-}deliver_p(m)) - ts(A\text{-}broadcast_r(m)) \geq ts(R\text{-}deliver_q(m)) - ts(R\text{-}broadcast_r(m)) + 1$ (3). Let $l^{RB}_{m,q} = ts(R\text{-}deliver_q(m)) - ts(R\text{-}broadcast_r(m))$ (4), and $l^{AB}_{m,p} = ts(A\text{-}deliver_p(m)) - ts(A\text{-}broadcast_r(m))$ (5). Therefore, from (3), (4), and (5), $l^{AB}_{m,p} \geq l^{RB}_{m,q} + 1$. From the definition of delivery latency, $dl^R(m) \geq l^{AB}_{m,p}$. It follows that $dl^R(m) \geq l^{AB}_{m,p} \geq l^{RB}_{m,q} + 1$. From Proposition 4.7, $l^{RB}_{m,q} \geq 1$, and we conclude that $dl^R(m) \geq 2$. □

Proposition 5.7 *Assume that Algorithm 2 uses the Reliable Broadcast implementation given in [CT96]. There is a run R generated by Algorithm 2 where message m is A-delivered during stage $k > 0$, and $dl^R(m) = 2$.*

PROOF. Immediate from Figure 5.10, where process p A-broadcasts message m. (Some messages have been omitted from Figure 5.10 for clarity.) Let $\rho, \rho' \in \{p, q, r, s\}$. It follows that $ts(receive_\rho(m)) = ts(send_p(m)) + 1$, and $ts(receive_\rho(k, \langle m \rangle) \text{ from } \rho') = ts(send_{\rho'}(k, \langle m \rangle)) + 1$. But $ts(send_{\rho'}(k, \langle m \rangle)) = ts(receive_{\rho'}(m))$, and therefore, $ts(receive_\rho(k, \langle m \rangle) \text{ from } \rho') = ts(send_p(m)) + 2$. From Figure 5.10, we have that $ts(A\text{-}broadcast_p(m)) = ts(send_p(m))$, and $ts(A\text{-}deliver_\rho(m)) = ts(receive_\rho(k, \langle m \rangle) \text{ from } \rho')$. By the definition of delivery latency, we conclude that $dl^R(m) = 2$. □

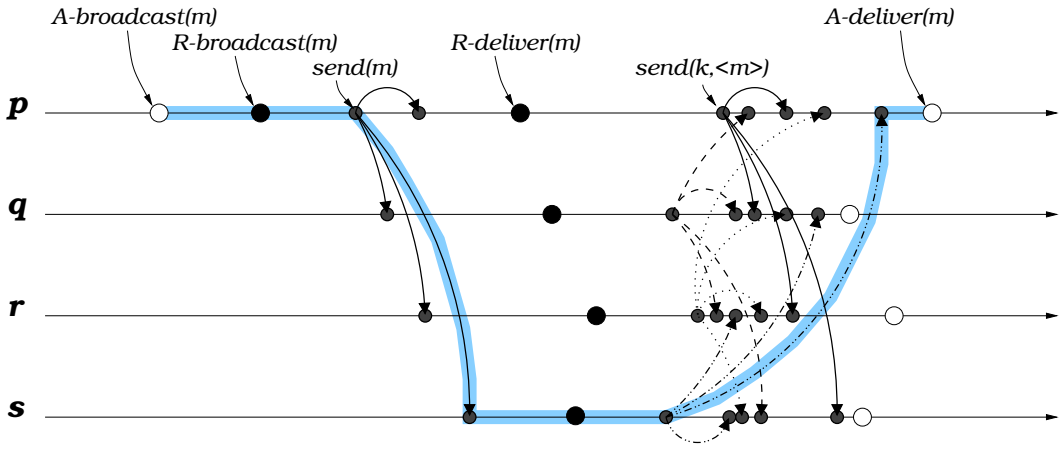

Figure 5.10: Run of OPT-ABcast with $dl^R(m) = 2$

The results that follow define the behaviour of the Optimistic Atomic Broadcast algorithm for messages A-delivered at the end of stage k. Proposition 5.8 establishes

a lower bound for this case, and Proposition 5.9 shows that this bound can be reached when there are no process failures and no failure suspicions.

Proposition 5.8 *Assume that Conjecture 4.1 is true (see page 72). There is no run R generated by Algorithm 2 where m and m' are the only messages A-delivered, m and m' are both A-delivered at line 30, and $dl^R(m) < 4$ and $dl^R(m') < 4$.*

Proof. Assume for a contradiction that there is a run R such that $dl^R(m) < 4$ and $dl^R(m') < 4$. Since p A-delivers m and m' at line 30, from Algorithm 2, p executes $decide_p(-, msgStg)$, such that $m, m' \in msgStg$. By uniform validity of Consensus, there is a process q that executes $propose_q(-, R\_delivered_q \ominus A\_delivered_q)$, such that $msgStg = R\_delivered_q \ominus A\_delivered_q$. Thus, q R-delivers message $(-, \text{EndStg})$. By uniform integrity of Reliable Broadcast, there is a process r that executes $R\text{-}broadcast_r(-, \text{EndStg})$. Therefore, r has R-delivered at least one message that is neither in $A\_delivered_r$ nor in $stgA\_deliver_r$ (line 13). Without loss of generality, assume that this message is m. Since r R-delivered m, there is a process s that executes $R\text{-}broadcast_s(m)$, and this is the process that executes $A\text{-}broadcast_s(m)$.

We define:

$$\begin{aligned}
l^{AB}_{m,p} &= ts(A\text{-}deliver_p(m)) - ts(A\text{-}broadcast_s(m)),\\
l^{C}_{p} &= ts(decide_p(-, msgStg)) - ts(propose_q(-, msgStg)),\\
l^{RB}_{\text{EndStg},q} &= ts(R\text{-}deliver_r(\text{EndStg})) - ts(R\text{-}broadcast_r(\text{EndStg})), \text{and}\\
l^{RB}_{m,r} &= ts(R\text{-}deliver_r(m)) - ts(R\text{-}broadcast_s(m)).
\end{aligned}$$

It follows that $l^{AB}_{m,p} \geq l^{C}_{p} + l^{RB}_{\text{EndStg},q} + l^{RB}_{m,r}$. From Conjecture 4.1 and the definition of latency degree [Sch97], $l^{C}_{p} \geq 2$, and from Proposition 4.7, $l^{RB}_{\text{EndStg},q} \geq 1$, and $l^{RB}_{m,r} \geq 1$. Thus, $l^{AB}_{m,p} \geq 4$. By the definition of delivery latency, $dl^R(m) \geq l^{AB}_{m,p}$, and we conclude that $dl^R(m) \geq 4$. □

Proposition 5.9 *Assume that Algorithm 2 uses the Reliable Broadcast implementation given in [CT96], and the Consensus implementation given in [Sch97]. There exists a run R of Algorithm 2 where messages m and m' are both A-delivered at line 30, and $dl^R(m) = 4$ and $dl^R(m') = 4$.*

Proof. Immediate from Figure 5.11, where process q A-broadcasts message m, and process r A-broadcasts message m'. (The Consensus execution and some messages have been omitted for clarity.) For all $\rho \in \{p, q, r, s\}$, $ts(receive_\rho(m)) = ts(send_q(m)) + 1$, and $ts(receive_\rho(m')) = ts(send_r(m')) + 1$. It also follows that $ts(receive_\rho(k, \text{EndStg})) = ts(send_s(k, \text{EndStg})) + 1$. From Figure 5.11, $ts(send_s(k, \text{EndStg})) = ts(receive_s(m)) = ts(receive_s(m'))$, and therefore, $ts(receive_\rho(k, \text{EndStg})) = ts(send_{\rho'}(m)) + 2$, $\rho' \in \{q, r\}$.

By the Consensus algorithm given in [Sch97], $ts(decide_\rho(-)) = ts(propose_\rho(-)) + 2$. From Figure 5.11, $ts(propose_\rho(-)) = ts(receive_\rho(k, \text{EndStg}))$, and we have that

$ts(decide_\rho(-)) = ts(receive_\rho(k, \text{ENDSTG})) + 4$. We conclude by the definition of delivery latency and since $ts(A\text{-}deliver_\rho(m)) = ts(A\text{-}deliver_\rho(m')) = ts(decide_\rho(-))$, $ts(A\text{-}broadcast_q(m)) = ts(send_q(m))$, and $ts(A\text{-}broadcast_r(m)) = ts(send_r(m))$, that $dl^R(m) = 4$ and $dl^R(m') = 4$. □

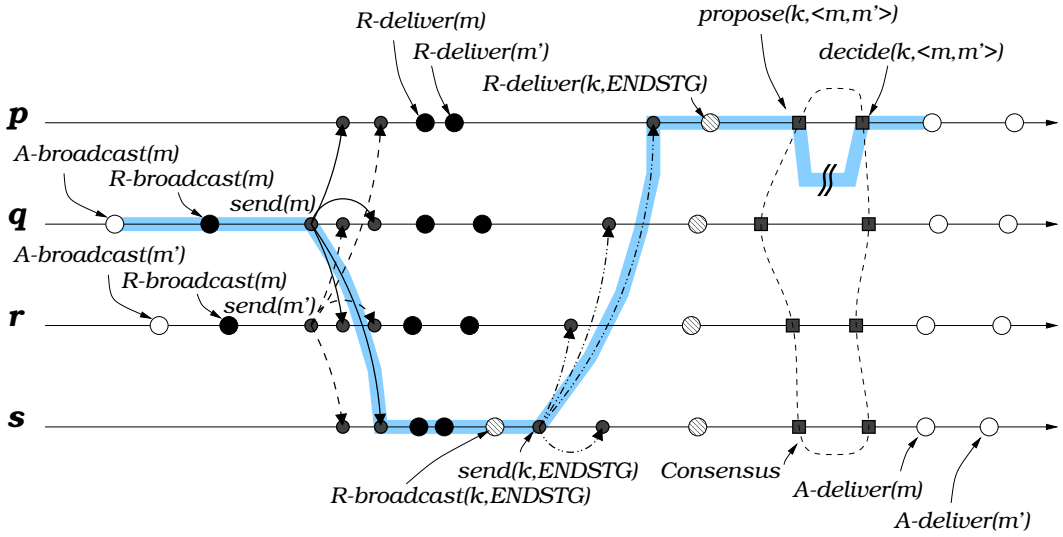

Figure 5.11: Run of OPT-ABcast with $dl^R(m) = 4$ and $dl^R(m') = 4$

5.3.3 Handling Failures

In the OPT-ABcast algorithm (line 18), whenever task $StgDeliver^k$ does not receive messages from all processes in Π, the current stage k is terminated, which leads to an execution of Consensus to A-deliver the messages. Therefore, as soon as a process $p \in \Pi$ crashes, the A-deliver of messages will always be *slow* (i.e., with a delivery latency of at least 4). This can be solved by adding a membership service to our OPT-ABcast algorithm as follows. Let v_i be the current view of system Π ($v_i \subseteq \Pi$):

- at line 18, replace condition $\pi = \Pi$ by $\pi = v_i$.

Once a process p crashes (or is suspected to have crashed), p is removed from the view and *fast* A-deliver of messages is again possible. We do not discuss further this extension to the OPT-ABcast algorithm, but we note that the instance of the membership problem needed to remove a crashed process can easily be integrated into the Consensus problem that terminates a stage.

5.4 Related Work

The work presented in this chapter combines Atomic Broadcast algorithms with optimistic techniques. The literature on Atomic Broadcast algorithms is abundant

(e.g., [AMMS+93, BSS91, CT96, CM84, GMS91, Jal98, LG90, WS95]). However, the multitude of different models and assumptions needed to prove the correctness of the algorithms renders any fair comparison difficult. We base our solution on the Atomic Broadcast algorithm of [CT96] because it provides a theoretical framework that permits to develop the correctness proofs under assumptions that are realistic in many settings (i.e., unreliable failure detectors).

Optimistic algorithms have been widely studied in database concurrency control (see Chapter 2). However, there have not been attempts, prior to this work, to introduce optimism in the context of agreement algorithms. The Classical Atomic Broadcast Algorithm with Optimistic Treatment approach described in Section 5.1 is conceptually similar to Virtual Time, and its implementation Time Warp [Jef85]. The Time Warp mechanism executes operations in a pre-determined virtual time. All operations have to be executed according this time, but since a process is never sure whether it has received all the operations that precede a given operation, in order to guarantee the order constraint, some previously operations may have to be undone, and processed again. An important difference between these two optimistic strategies is that operations are undone and re-executed at most once with the Classical Atomic Broadcast Algorithm with Optimistic Treatment approach, but an unbounded number of times with the Time Warp mechanism.

The closest to the idea exploited by the Optimistic Atomic Broadcast algorithm is [GLS96], where the authors reduce the Atomic Commitment problem to Consensus and, in order to have a fast decision, exploit the following property of the Consensus problem: if every process starts Consensus with the same value v, then the decision is v. This work presents a more general idea, and does not require that all the initial values be equal. Moreover, we have here the trade-off of typical optimistic algorithms: if the optimistic assumption is met, there is a benefit (in efficiency), but if the optimistic assumption is not met, there is a loss (in efficiency).

5.5 Discussion

The work presented in this chapter originated from the pragmatic observation that, with high probability, messages broadcast in a local area network are "spontaneously" totally ordered. Exploiting this observation led to proposing the optimistic approaches, and developing the Optimistic Atomic Broadcast algorithm. Processes executing the OPT-ABcast algorithm progress in a sequence of stages, and messages can be delivered during stages or at the end of stages. Messages are delivered faster during stages than at the end of stages. For any process, the current stage is terminated, and another one started, whenever the spontaneous total order property does not hold.

The efficiency of the OPT-ABcast algorithm has been quantified using the notion of delivery latency. The delivery latency of messages delivered during a certain stage has been shown to be equal to 2, while the delivery latency of messages delivered at the end of a stage equal to 4. This result shows the tradeoff of the OPT-ABcast algorithm: if most messages are delivered during the stages, the OPT-ABcast algo-

rithm outperforms known Atomic Broadcast algorithms, otherwise, the OPT-ABcast algorithm is outperformed by known Atomic Broadcast algorithms.

Finally, to the best of our knowledge, there have not been previous attempts of exploiting optimistic properties for the development of agreement algorithms. If this property is satisfied the efficiency of the algorithm is improved, if the property is not not satisfied the efficiency of the algorithm deteriorates (however the optimistic property has no impact on the safety and liveness guarantees of the system). We believe that this opens interesting perspectives for revisiting or improving other agreement algorithms.

Chapter 6

Conclusion

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

Distributed computing has enabled the development of applications and services that were not feasible before computers started to communicate to each other. Several current applications show evidence that the distributed computing paradigm is reshaping the way people think about and do daily life activities. Consequently, the dissemination of distributed applications is increasing the demand for high-availability and high-performance mechanisms to support these applications.

However, designing high-availability systems that provide good performance has been the hole grail of fault tolerant computing. In order to reach this objective, some proposals in the context of database systems have suggested weakening consistency guarantees. This approach is very attractive in some cases, but to be effective, deep knowledge about the application is usually necessary. More recently, some researchers have proposed to use group communication mechanisms to develop high-availability and high-performance databases that also ensure strong data consistency.

This thesis discusses the details involved in the design of a replicated database protocol based on group communication primitives, and proposes the use of application semantics and optimistic techniques to develop efficient group communication primitives.

6.1 Research Assessment

This research has led to four major contributions. In the database domain, the Database State Machine and the Reordering technique have been proposed. In the distributed system domain, the Generic Broadcast problem and algorithm and the

Optimistic Atomic Broadcast algorithm have been introduced.

Database State Machine. This thesis has presented the Database State Machine, an approach to executing transactions in a cluster of database sites that communicate by message passing, and do not have access to shared memory nor to a common clock. In the Database State Machine, read-only transactions are processed locally on a database site, and update transactions do not incur in any synchronisation among sites during their execution. When an update transaction requests a commit operation, it is atomically broadcast. Local execution of update transactions on database sites can be seen as a *pre-processing*, since a transaction can only be committed (i.e., updates applied to the database) by some sites after the transaction is delivered and successfully certified on this database site. Consistency is guaranteed by a local concurrency control mechanism (two phase locking), and the certification test.

Some important aspects about the Database State Machine are that transactions are never involved in distributed deadlocks (only local deadlocks are possible), the load can be fairy distributed in the system (local transactions are executed locally and update transactions are pre-processed by only one database site), and all communication is encapsulated in the Atomic Broadcast primitive. Basing all database site interaction on a high level group communication primitive has some benefits. First, it simplifies the portability of the Database State Machine to systems with different network characteristics (i.e., only the Atomic Broadcast primitive has to be re-implemented). Second, it focuses efforts to improve communication performance on a single point, and finally, it simplifies the proof of correctness of the protocol.

Reordering Technique. The certification test necessary to commit an update transaction is an optimistic way of processing transactions. Depending on the profile of the transactions (e.g., number of read and write operations), and characteristics of the database (e.g., number of data items), optimistic concurrency control may result in high abort rates. In order to increase the number of transactions that pass the certification test, we have introduced the Reordering technique. The Reordering technique originated from the observation that concurrent transactions can be certified in any order, but since some orders can lead to more aborts than others, instead of taking a chance, the Reordering technique looks for favourable certification orders. Simulation results show that this can be very effective.

The Reordering technique was implemented in the Database State Machine by means of a Reorder List with maximum size determined by the Reorder Factor. At first glance, the greater the Reorder Factor, the better. Nevertheless, big Reorder Factors have the undesirable side effect of augmenting the system's response time. Therefore, a "good" Reorder Factor is a compromise between abort rate and response time, and depends on system characteristics. The Reorder Factor allows the Database State Machine to be tuned according to the system requirements.

Generic Broadcast. So far, order properties offered by group communication primitives existed in two flavours: no message order guarantee and message order guarantee for all messages.[1] Such primitives, Reliable and Atomic Broadcast, respectively, are important abstractions, however, in several scenarios, Reliable Broadcast is too weak to ensure system correctness, and Atomic Broadcast is to strong. Since ordering messages has a cost, to be efficient, applications need group communication with order guarantees that match their exact necessities. This observation was the starting point for the conception of Generic Broadcast.

Generic Broadcast permits an application to define any order semantics that it needs. In addition to defining Generic Broadcast, we have also proposed an algorithm that solves it. The Generic Broadcast algorithm proposed uses a quorum to determine when messages can be safely delivered without the cost of a Consensus execution (whose aim is to order messages), and when messages conflict, and so, Consensus is necessary. No previous attempt of defining a primitive like Generic Broadcast is known. When messages do not conflict, the Generic Broadcast algorithm has a smaller delivery latency than known Atomic Broadcast algorithms, and when messages conflict, it has a delivery latency greater than the delivery latency of known Atomic Broadcast algorithms.

Optimistic Atomic Broadcast. We have described three optimistic approaches in the context of Atomic Broadcast. These approaches take advantage of the spontaneous total order property, typical in local area networks. The approaches based on optimistic treatment guarantee different properties from Atomic Broadcast. We have also presented in detail an Optimistic Algorithm with Conservative Treatment, the Optimistic Atomic Broadcast algorithm.

A very simple analysis shows that the approaches based on optimistic treatment outperform the Optimistic Atomic Broadcast with Conservative Treatment approach. Nevertheless, the former two allow messages to be delivered twice, and so, they cannot replace Atomic Broadcast without changes in the application, which is possible with the Optimistic Atomic Broadcast with Conservative Treatment approach. Therefore, applications using the approaches based on optimistic treatment must be able to cope with messages delivered first in a tentative order that may be different from a definitive order [KPAS99].

6.2 Future Directions and Open Questions

Besides the contributions presented in the previous section, this work has raised several issues that deserve further analysis. In the following, we describe some future directions and open questions related to this research.

[1] This includes Total Order and Causal Order. Only Total Order has been considered in the thesis.

Safety vs. Liveness Database Guarantees. Traditionally, database protocols have only been concerned with safety properties (i.e., ACID properties) [BHG87, GR93], and very few works have addressed liveness properties (e.g., [RSL78, PG97]). The Database State Machine could be used as a framework to study liveness guarantees in replicated databases. The fact that Atomic Broadcast is defined by safety and liveness guarantees may help characterise the liveness property ensured by the Database State Machine. As a second step, it would be interesting to study how to define and achieve stronger and weaker liveness guarantees.

The Database State Machine in Practice. Simulation results have brought to light some of the characteristics of the Database State Machine. Experiments using a "real setting" would be interesting to take further conclusions about the approach. The Database State Machine was designed in such a way to simplify its integration with existing database engines (e.g., without modifying internal code). Some preliminary studies involving the POET database [Obj97] have shown that the Database State Machine can indeed be integrated in an existing database engine without changing the database engine's code. However, additional work is still necessary to conceive a prototype.

Partial Replication. The Database State Machine assumes that each database site has a full copy of the database. This hypothesis allows database sites to execute the certification test independently of one another, and reach the same outcome. One natural question is whether it is possible to build a Database State Machine based on a weaker assumption (i.e., partial replication). It seems that this can only be done by introducing some coordination among database sites, when executing the certification test. The resulting protocol would be a sort of Atomic Commitment.

One might wonder whether total order is still necessary in this scenario. It turns out that the answer is affirmative, since it has been shown that if database sites certify transactions in the same order, the certification test can be much more effective (i.e., more transactions pass the test) [PGS98]. The exact way transactions are executed, broadcast, and certified in this scenario is subject to further investigation.

Group Communication in the Crash-Recover Model. Only recently, group communication in the (asynchronous) crash-recover model has attracted the attention of researchers. Works developed so far have focused on solving Consensus in the crash-recover model [OGS97, HMR97, ACT98]. This is an important step towards group communication protocols in the crash-recover model since some group communication problems have been shown to be equivalent to Consensus (e.g., Atomic Broadcast). Although these results were developed in the crash-stop model, it is reasonable to expect that they have analogues in the crash-recover model. To the present time, no work has explicitly addressed the problem of group communication in the crash-recover model where all processes can crash and recover.[2] This seems

[2]Some group communication toolkits allow new processes to join processes in execution [BJ87, Mal96, vBM96]. This mechanism can be seen as a kind of "recover," since a process that has

to be a fruitful research direction for the next years.

Optimistic Generic Broadcast. The ideas underlying Generic Broadcast and Optimistic Atomic Broadcast are orthogonal, and one could think of combining them. The result would be an optimistic implementation of Generic Broadcast. For example, the Optimistic Generic Broadcast algorithm would only order messages if they conflict and the spontaneous total order property does not hold. Such mechanism would reduce the likelihood that messages have to be ordered with a Consensus.

The Optimistic Design Principle. Some thoughts about the Optimistic Atomic Broadcast algorithm suggest an *optimistic design principle*. The idea is that in some circumstances, a problem can be solved by two mechanisms: a fast mechanism, that ensures the problem properties in most cases, and a slow mechanism, that always guarantees the problem properties. By being able to detect whenever the first mechanism does not succeed, and switch to the second whenever this happens, a system designer can come up with an optimistic way of solving a problem. This optimistic design principle requires refinements, according to the situation where it is applied. For example, in same cases, wrong results produced by the fast mechanism should never be observed by the application, and in other cases, this may be tolerated. The study about degrees of optimism shows that the optimistic design principle can be put in practice in both cases. Furthermore, while the implementation of the fast and the slow mechanisms depend on specific characteristics about the problem being solved and the model, the detection mechanism might exploit research done on the detection of global predicates [CL85, BM93].

crashed can restart again (with a different identification). Nevertheless, correctness is not ensured if all processes crash and then recover.

Bibliography

[AAAS97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), September 1997.

[AAS97] D. Agrawal, A. El Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson (USA), May 1997.

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

[ACT97] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG'97)*, pages 126–140, Saarbrücken (Germany), September 1997.

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the International Symposium on Distributed Computing (DISC'98)*, pages 231–245, September 1998.

[ADMSM94] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and efficient replication using group communication. Technical Report CS94-20, Hebrew University of Jerusalem, 1994.

[AM92] E. Anceaume and P. Minet. Etude de protocoles de diffusion atomique. Technical Report RR-1774, Inria, Institut National de Recherche en Informatique et en Automatique, October 1992.

[AMMS+93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, P. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh (USA), May 1993.

[BCBT96] A. Basu, B. Charron-Bost, and S. Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609, Cornell University, September 1996.

[Bha99] B. K. Bhargava. Concurrency control in database systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):3–16, January 1999.

[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 11th ACM Symposium on OS Principles*, pages 123–138, November 1987.

[BK97] Y. Breitbart and H. F. Korth. Replication and consistency: being lazy helps sometimes. In *Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 173–184, New York (USA), 1997.

[BK98] G. Alonso B. Kemme. Database replication based on group communication. Technical Report 289, ETH Zurich, Department of Computer Science, 1998.

[BM93] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

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

[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.

[CHTCB96] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 322–330, New York (USA), May 1996.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM84] J. M. Chang and N. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Cri91] F. Cristian. Understanding fault-tolerant distributed Systems. *Communications of the ACM*, 34(2):56–78, February 1991.

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

[Del95] D. J. Delmolino. Strategies and techniques for using Oracle 7 replication. Technical report, Oracle Corporation, 1995.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver (Canada), August 1987.

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

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

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles SOSP 7*, pages 150–162, Pacific Grove (USA), December 1979.

[GLS96] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 692–697, Hong Kong, May 1996.

[GMS91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[Gol95] R. Goldring. Update replication: What every designer should know. *Info DB*, 9(2), April 1995.

[GOS96] R. Guerraoui, R. Oliveira, and A. Schiper. Atomic updates of replicated data. In *Proceedings of the European Dependable Computing Conference*, Taormina (Italy), 1996.

[GR93] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Gro98] Object Management Group. Fault tolerant CORBA using entity redundancy. Request for proposal, Object Management Group, Framingham Corporate Center, Framingham (USA), April 1998.

[GS97] R. Guerraoui and A. Schiper. Software based replication for fault tolerance. *IEEE Computer*, 30(4), April 1997.

[Gue95] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG'95)*, pages 87–100, Le Mont Saint Michel (France), September 1995.

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

[Her86] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[HMR97] M. Hurfin, A. Mostefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. Technical report, Institut de Recherche en Informatique et Système Aléatoires, Université de Rennes, 1997.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[Jac95] K. Jacobs. Concurrency control, transaction isolation and serializability in SQL92 and Oracle 7. Technical report, Oracle Corporation, July 1995.

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

[Jaj99] S. Jajodia. Data replication gaining popularity. *IEEE Concurrency*, pages 85–86, April–June 1999.

[Jal98] P. Jalote. Efficient ordered broadcasting in reliable csma/cd networks. In *Proceeding of the 18th International Conference on Distributed Computing Systems*, pages 112–119, Amsterdam (The Netherlands), May 1998.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[JM87] S. Jajodia and D. Mutchler. Dynamic voting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 227–238, May 1987.

[JMR97] H. V. Jagadish, I. S. Mumick, and M. Rabinovich. Scalable versioning in distributed databases with commuting updates. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, pages 520–531, April 1997.

[Kei94] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem (Israel), 1994.

[KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, Austin (USA), June 1999.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LG90] S. W. Luan and V. D. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):271–285, July 1990.

[Mal96] C. P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, September 1996.

[MR99] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach. In *Proceeding of the International Symposium on Distributed Computing (DISC'99)*, pages 49–63, September 1999.

[Obj97] Object Database Management Group. *POET Java SDK Programmer's Guide*, 1997.

[OGS97] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report TR-97/239, EPFL, Computer Science Department, August 1997.

[Ora95] Oracle Corporation. *Oracle 7 Distributed Database Technology and Symmetric Replication*, April 1995.

[OV99] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 2nd edition, 1999.

[Pap79] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[Ped97] F. Pedone. A closer look at optimistic replica control. In *Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, Zinal (Switzerland), March 1997.

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

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

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

[PS98] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98, formerly WDAG)*, September 1998.

[PS99a] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, September 1999.

[PS99b] F. Pedone and A. Schiper. Optimistic atomic broadcast: A pragmatic viewpoint. Technical Report SSC/1999/015, École Polytechnique Fédérale de Lausanne, Switzerland, May 1999.

[RL92] S. Ghemawat R. Ladin, B. Liskov. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[RSL78] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[SA93] O. T. Satyanarayanan and D. Agrawal. Efficient execution of read-only transactions in replicated multiversion databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):859–871, October 1993.

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

[Sch93] F. B. Schneider. What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems*, chapter 2. Addison-Wesley, 2nd edition, 1993.

[Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[Ske81] D. Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 133–142, Ann Arbor (USA), April 1981.

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

[Sta95] D. Stacey. Replication: DB2, Oracle, or Sybase? *SIGMOD Record*, 24(5):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.

[Syb99] Sybase. *Sybase SQL Anywhere and Replication Server: the enterprise-wide replication solution*, 1999.

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

[Tho98] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):173–189, January 1998.

[TTP+95] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. H. Hauser. Managing update conflict in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort (USA), December 1995.

[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.

[WPS99] M. Wiesmann, F. Pedone, and A. Schiper. A systematic classification of replicated database protocols based on atomic broadcast. In *Proceedings of the 3rd Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira (Portugal), April 1999.

[WS95] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS'14)*, pages 106–115, Bad Neuenahr (Germany), September 1995.

Appendix A

Broadcasts and Consensus Algorithms

This appendix presents broadcast and Consensus algorithms referenced throughout this thesis. The Reliable Broadcast (Algorithm 3), Consensus (Algorithm 4), and Atomic Broadcast (Algorithm 6) algorithms have been proposed by Chandra and Toueg [CT96]. The Early Consensus algorithm (Algorithm 5) has been proposed by Schiper [Sch97]. All algorithms assume the asynchronous model augmented with failure detectors where processes communicate by message passing, using reliable channels, and fail by stopping their computation (i.e., same model as the one considered in Chapters 3 and 4). The Consensus algorithms use a failure detector of class $\diamond S$.

Algorithm 3 Reliable broadcast algorithm

Every process p executes the following:

To execute *R-broadcast*(m):

  send(m) *to all (including p)*;

R-deliver(m) occurs as follows:

  when *receive*(m) for the first time
    if *sender*(m) $\neq p$ **then** *send*(m) *to all*;
    R-deliver(m);

Algorithm 4 Chandra and Toueg consensus algorithm

Every process p executes the following:

procedure *propose*(v_p)
  $estimate_p \leftarrow v_p$
  $state_p \leftarrow undecided$
  $r_p \leftarrow 0$
  $ts_p \leftarrow 0$

  while $state_p = undecided$ **do**
    $r_p \leftarrow r_p + 1$
    $c_p \leftarrow (r_p \bmod n) + 1$

    send $(p, r_p, estimate_p, ts_p)$ to c_p *{**Phase 1**}*

    if $p = c_p$ **then** *{**Phase 2**}*
      wait until for $\lceil (n+1)/2 \rceil$ processes q: received$(q, r_p, estimate_q, ts_q)$ from q]
      $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
      $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
      $estimate_q \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
      send $(p, r_p, estimate_q)$ to all

    wait until [received $(c_p, r_p, estimate_q)$ from c_p **or** $c_p \in \mathcal{D}_p$] *{**Phase 3**}*
    if [received $(c_p, r_p, estimate_q)$ from c_p] **then**
      $estimate_q \leftarrow estimate_{c_p}$
      $ts_p \leftarrow r_p$
      send (p, r_p, ack) to c_p
    else
      send $(p, r_p, nack)$ to c_p

    if $p = c_p$ **then** *{**Phase 4**}*
      wait until [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes q : received (q, r_p, ack) **or** $(q, r_p, nack)$]
      if [for $\left\lceil \frac{(n+1)}{2} \right\rceil$ processes q : received (q, r_p, ack)] **then**
        R-broadcast$(p, r_p, estimate_p, decide)$

when *R-deliver*$(q, r_q, estimate_q, decide)$
  if $state_p = undecided$ **then**
    $decide(estimate_q)$
    $state_p \leftarrow decided$

Algorithm 5 Early consensus algorithm

```
function propose(v_p)
  r_i ← 0
  estimate_i ← (i, v_i)
cobegin
  upon reception of (p_j, r_j, v_j, decide) from p_j:
    send(p_i, r_j, v_j, decide) to all;
    return v_j

  loop
    phase_i ← 1; currentRoundTerminated_i ← false;
    coordSuspected_i ← false; nbSuspicions_i ← 0;
    coord_i ← (r_i mod n) + 1;

    if i = coord_i then
      send (p_i, r_i, 1, estimate_i) to all;

    while not currentRoundTerminated_i
      select
        upon reception of (p_j, r_i, 1, estimate_j) from p_j when phase_i = 1:
          first reception:
            msgCounter_i ← 1;
            if i ≠ coord_i then
              estimate_i ← estimate_j;
              send (p_i, r_i, 1, estimate_i) to all;
          other receptions:
            msgCounter_i ← msgCounter_i + 1;
          if msgCounter_i > n/2 then
            send (p_i, r_i, estimate_i.second, decide) to all;
            return estimate_i.second;

        upon coord_i ∈ ◇S_i when not coordSuspected_i:
          send (p_i, r_i, suspicion) to all;
          coordSuspected_i ← true;

        upon reception of (p_j, r_i, suspicion) from p_j when phase_i = 1:
          nbSuspicions_i ← nbSuspicions_i + 1;
          if nbSuspicions_i > n/2 then
            phase_i ← 2;
            send (p_i, r_i, 2, estimate_i) to all;

        upon reception of (p_j, r_i, 2, estimate_j) from p_j:
          first reception:
            msgCounter_i ← 1;
            if phase_i = 1 then
              phase_i ← 2;
              send (p_i, r_i, 2, estimate_i) to all;
          other receptions:
            msgCounter_i ← msgCounter_i + 1;
          if estimate_j.first = coord_i then estimate_i ← estimate_j;
          if msgCounter_i > n/2 then
            currentRoundTerminated_i ← true;
            r_i ← r_i + 1;
            estimate_i.first ← i;
```

Algorithm 6 Atomic broadcast algorithm

Every process p executes the following:

Initialisation:
  $R\_delivered \leftarrow \epsilon$
  $A\_delivered \leftarrow \epsilon$
  $k \leftarrow 0$

To execute *A-broadcast*(m): ***{Task 1}***

  R-broadcast(m)

A-deliver$(-)$ occurs as follows:

  when *R-deliver*(m) ***{Task 2}***
    $R\_delivered \leftarrow R\_delivered \cup \{m\}$

  when $R\_delivered \setminus A\_delivered \neq \emptyset$ ***{Task 3}***
    $k \leftarrow k + 1$
    $A\_undelivered \leftarrow R\_delivered \setminus A\_delivered$
    propose$(k, A\_undelivered)$
    wait until *decide*$(k, msgSet^k)$
    $A\_deliver^k \rightarrow msgSet^k \setminus A\_delivered$
    atomically deliver all messages in $A\_deliver^k$ in some deterministic order
    $A\_delivered \leftarrow A\_delivered \cup A\_deliver^k$

Curriculum Vitae

I was born in Rio Grande, Rio Grande do Sul, south of Brazil, in 1969. From 1974 to 1985 I attended primary and secondary schools. My passion for computers started back in the early eighties, when I did my first "empirical" experiments with a Sinclair (actually, a Z80 with 16 KBytes of RAM, which I still keep). From 1986 to 1989 I had a more formal introduction to Computer Science at Colégio Técnico Industrial, a technical high school in Rio Grande. In 1992, I graduated as a Mechanical Engineering from University of Rio Grande. In my last years at the university, I taught Computer Science at Colégio Técnico Industrial.

From 1992 to 1994 I moved to Porto Alegre, capital of Rio Grande do Sul to deepen my knowledge about computers through a Master Science. During my Master I studied, among other things, distributed systems, databases, and software based fault tolerant systems. In mid-1995 I moved to France where I spent one year. Apart from getting in close contact with the French language and culture, I continued my research work at INRIA Grenoble. In September 1996 I started my Ph.D at EPF Lausanne, Switzerland. In the context of my doctorate, a joint effort between the Swiss Federal Institute of Technology at Lausanne and Zurich, I had the opportunity to work with several people, give research talks (at interesting places), and interact with industrial partners.