AGREEMENT-RELATED PROBLEMS: FROM SEMI-PASSIVE REPLICATION TO TOTALLY ORDERED BROADCAST

THÈSE N^{O} 2229 (2000)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Xavier DÉFAGO

Ingénieur informaticien diplômé EPF

de nationalités française et suisse (origine Val-d'Illiez, VS)

acceptée sur proposition du jury:

Prof. A. Schiper, directeur de thèse

Prof. J.-Y. Le Boudec, rapporteur

Prof. D. Malkhi, rapporteur

Prof. F. Mattern, rapporteur

Lausanne, EPFL

2000

Abstract

Agreement problems constitute a fundamental class of problems in the context of distributed systems. All agreement problems follow a common pattern: all processes must agree on some common decision, the nature of which depends on the specific problem. This dissertation mainly focuses on three important agreements problems: *Replication*, *Total Order Broadcast*, and *Consensus*.

Replication is a common means to introduce redundancy in a system, in order to improve its availability. A replicated server is a server that is composed of multiple copies so that, if one copy fails, the other copies can still provide the service. Each copy of the server is called a replica. The replicas must all evolve in manner that is consistent with the other replicas. Hence, updating the replicated server requires that every replica agrees on the set of modifications to carry over. There are two principal replication schemes to ensure this consistency: *active* replication and *passive* replication.

In Total Order Broadcast, processes broadcast messages to all processes. However, all messages must be delivered in the same order. Also, if one process delivers a message m, then all correct processes must eventually deliver m.

The problem of Consensus gives an abstraction to most other agreement problems. All processes initiate a Consensus by proposing a value. Then, all processes must eventually decide the same value v that must be one of the proposed values.

These agreement problems are closely related to each other. For instance, Chandra and Toueg [CT96] show that Total Order Broadcast and Consensus are equivalent problems. In addition, Lamport [Lam78] and Schneider [Sch90] show that active replication needs Total Order Broadcast. As a result, active replication is also closely related to the Consensus problem.

The first contribution of this dissertation is the definition of the semi-passive replication technique. Semi-passive replication is a passive replication scheme based on a variant of Consensus (called Lazy Consensus and also defined here). From a conceptual point of view, the result is important as it helps to clarify the relation between passive replication and the Consensus problem. In practice, this makes it possible to design systems that react more quickly to failures.

The problem of Total Order Broadcast is well-known in the field of distributed systems and algorithms. In fact, there have been already more than fifty algorithms published on the problem so far. Although quite similar, it is difficult to compare these algorithms as they often differ with respect to their actual properties, assumptions, and objectives. The second main contribution of this dissertation is to define five classes of total order broadcast algorithms, and to relate existing

algorithms to those classes.

The third contribution of this dissertation is to compare the expected performance of the various classes of total order broadcast algorithms. To achieve this goal, we define a set of metrics to predict the performance of distributed algorithms.

Résumé

Les problèmes d'*accord* forment une classe fondamentale de problèmes dans le contexte des systèmes répartis. Les problèmes d'accord suivent un schéma commun: tous les processus doivent arriver à une décision commune, laquelle dépend de la nature exacte du problème. Cette thèse met principalement l'accent sur trois importants problèmes d'accord: la *réplication*, la *diffusion totalement ordonnée*, ainsi que le *consensus*.

Le *réplication* est une technique usuelle permettant d'introduire de la redondance dans un système, afin d'en garantir une meilleure disponibilité. Un serveur répliqué est un serveur qui est constitué de plusieurs copies afin que, si l'une d'entre elles tombe en panne, les autres puissent continuer à fournir le service. Les copies d'un serveur répliqué sont appelées des réplicas. Ces réplicas doivent évoluer de manière cohérente. Ainsi, la modification du serveur répliqué nécessite aux réplicas de se mettre d'accord sur l'ensemble des modifications à apporter à leurs états respectifs. On considère principalement deux approches générales permettant de maintenir cette cohérence: la réplication *active* et la réplication *passive*.

La *diffusion totalement ordonnée* est un autre problème d'accord. Des processus diffusent des messages à tous les processus avec la contrainte que tous les messages doivent impérativement être reçus dans le même ordre par tous les processus. De surcroît, si un processus reçoit un message m, alors le problème exige que tous les processus correct aient la garantie de recevoir le message m de manière inéluctable.

Le problème du *consensus* fourni une abstraction à la plupart des autres problèmes d'accord. Tous les processus débutent un consensus en proposant une valeur. Puis, tous les processus doivent finalement sélectionner la même valeur v, qui doit être la valeur proposée initialement par l'un des processus.

Ces problèmes d'accord sont liés les uns aux autres. Par exemple, Chandra et Toueg [CT96] ont montré que la diffusion totalement ordonnée et le consensus sont des problèmes équivalents. Lamport [Lam78] et Schneider [Sch90] ont aussi montré que la réplication active a besoin de la diffusion totalement ordonnée. Ceci signifie que la réplication active est aussi intimement liée au problème du consensus.

La première contribution de cette thèse a été de définir la technique de réplication semi-passive. Il s'agît d'une technique de réplication passive basée sur une variante du problème de consensus: *consensus paresseux* (Lazy Consensus; aussi définie dans ce document). Le résultat est tout d'abord important d'un point de vue conceptuel, car il permet d'éclaircir la relation qui éxiste entre la réplication passive et le problème du consensus. D'un point de vue pratique, ceci permet

de concevoir des systèmes qui réagissent plus rapidement en cas de panne.

Le problème de la diffusion totalement ordonnée est bien connu dans le domaine des systèmes et de l'algorithmique répartis. A tel point que plus d'une cinquantaine d'algorithmes on été publiés jusqu'à présent. Malgré leurs similarités, il est difficile de comparer ces algorithmes tant ils diffèrent par leurs propriétés, leurs hypothèses et leurs objectifs. La deuxième contribution de cette recherche a consisté à définir cinq classes d'algorithmes de diffusion totalement ordonnée, et de rattacher les algorithmes éxistants à ces classes.

La troisième contribution de cette recherche a été de comparer les performances supposées des diverses classes d'algorithmes de diffusion totalement ordonnée. Afin d'atteindre cet objectif, il a été nécessaire de définir un ensemble de métriques permettant de prédire les performances d'algorithmes répartis.

To those I love...
You know who You are.

Acknowledgments

First of all, I want to express my gratitude to Professor André Schiper for guiding my first steps into academic research. André has taught me so many things that I am unable to mention them all here. Among those I should mention a rigorous approach to problems and a strong research ethics. Moreover, I would like to thank André very deeply for his trust when I went to Japan to work on the material of Chapters 3 and 4.

I am also very grateful to Péter Urbán for his helpful collaboration and support at the most time-critical period of this research. More specifically, Péter has largely contributed to the material that is presented in Chapter 4 to Chapter 6, and has helped improve the overall presentation with his numerous comments.

Reviewing and reporting on a Ph.D. dissertation takes time and a lot of efforts. For this, I would like to thank the members of the jury, Professor Jean-Yves Le Boudec, Professor Dahlia Malkhi, and Professor Friedemann Mattern, as well as the president of the jury, Professor Roger David Hersch.

I would also like to thank those who have accepted to proofread this dissertation, thus making it possible to dramatically improve its content: Bernadette Charron-Bost, Fernando Pedone, Roel Vandewall, and Matthias Wiesmann.

My deep thanks go to the people who contributed, through their comments and suggestions, to various parts of this research. Bernadette Charron-Bost for many constructive critics about the description of system models, process controlled crash, and Atomic Broadcast. Jean-Yves Le Boudec for his suggestions that led to define the metrics presented in Chapter 5. Fernando Pedone for his numerous comments on the specification of replication techniques as well as many other aspects of this dissertation.

I want to thank all members of the LSE, past and present, for their support and friendship—in particular, Pascal Felber for helping me dive into CORBA when I joined the lab, Rachid Guerraoui for convincing me to come to the LSE in the first place, Rui Oliveira for sparking my interest in distributed algorithms, Fernando Pedone for initiating me to such an esoteric activity as the writing of proofs, Nicole Sergent for the numerous brainstorming sessions that paved the road to develop semi-passive replication, Kristine Verhamme for helping me out of the overwhelming administrative hassles, Matthias Wiesmann for his constructive comments and for always being so helpful, Uwe Wilhelm for giving me the opportunity to bring my modest contribution to his ideal of privacy. Also a special thanks to those with whom I have shared my teaching activities over the years—most notably, Philippe Altherr, Fernando Pedone, Michel Schinz, Péter Urbán, Roel

Vandewall, and Matthias Wiesmann.

Finally, I want to express my love and gratitude to my family and close friends. Their constant support and encouragements were determinant for the completion of this dissertation. I am especially grateful to my parents for providing most of the logistic support. Also, I want to thank my wife Yumi for teaching me how to always see the positive aspects of life. Last but not least, I would like to thank my daughter Yuki for her tenderness, and also for her generous contribution to the writing of this dissertation: "*gbbgggggggggg*."

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

There is nothing permanent except change.

— **Heraclitus of Ephesus** (ca. 525–475 B.C.)

Nowadays, computers are seldom the stand-alone machines that they used to be. The incredible evolution of computer networking in the last two decades has generalized the idea that computers work better together. Increasingly faster and cheaper networks, improved interaction models (e.g., WWW) combined with an ever-growing demand for large-scale communication, have contributed to the widespread acceptance of wide area networks (WANs), such as the Internet. Moreover, the ability to communicate is not limited to desktop computers alone. This is shown by the huge market of embedded systems, home appliances, personal digital assistants (PDAs) for which connectivity is becoming a prerequisite rather than a feature.

Distributed systems are based on the idea that independent computers can communicate in order to execute coordinated actions. Agreement is a special form of coordination. Agreement problems constitute a class of problems that consist in reaching a common decision between distributed entities. As such, agreement problems lie at the heart of complex distributed systems and applications. This dissertation is centered around agreement problems, and mainly focuses on three specific problems: *Replication*, *Total Order Multicast*, and indirectly *Consensus*.

1.1 Replicated Services

Considering that more and more people rely on remote services, there is a strong incentive to improve the reliability of those services. This is especially true when one considers the potential losses that failures can cause, such as human lives, money, or reputation. Laprie [Lap92] has formalized a set of parameters to evaluate the dependability of applications and systems. In this dissertation, we are interested in two of those parameters, namely the reliability and the availability of a system. The *reliability* of a system is defined as the mean time between two consecutive failures, while its *availability* is defined as the probability that, at any given time, the system is functioning.

1.1.1 Replication Techniques

Replication techniques are a means to replicate critical components of system, thus introducing redundancy. Replication is actually the only mechanism for fault tolerance that simultaneously increases the reliability and the availability of a system. The application model that is mainly considered in this dissertation is the client-server model. A client program issues requests to remote services. These services process the request, possibly modify their internal state, and send back a reply to the client. In a replicated service, there exist multiple copies of the service, called *replicas*, and the system must ensure that the internal state of each replica remains consistent with that of the others. There exist two general techniques for replication; active and passive replication.

Active Replication

In active replication, also called the state machine approach [Sch90, Sch93, Pol94], all replicas perform exactly the same actions in the same order, thus remaining identical. The major advantage of active replication is that it keeps a good response time in case of failures. However, it also requires that all actions are performed in a deterministic manner—an action is *deterministic* if its result only depends on the request that started it and the current state of the replica. The key problem in the implementation of active replication lies in the ability to impose a total order on request messages. Hence, in distributed systems, active replication relies on solving a problem called Total Order Multicast which guarantees that all messages sent to a set of destinations are delivered by those destinations in the same relative order, even in the presence of failures.

Passive Replication

In passive replication, also called the primary-backup approach [BMST93], only one replica (primary) executes the actions and updates the other replicas (backups). This technique requires to (1) select a primary, and (2) update the backups properly. Passive replication allows non-deterministic actions to be performed, and incurs less processing power than active replication. Despite its advantages, passive replication is commonly reproached to sometimes show extremely bad performance in case of failures.

1.1.2 Reaction to Failures

Most distributed applications that are developed nowadays are intrinsically reactive, and thus put some performance constraints on the responsiveness of the application. Reactive applications are often considered as a class of soft real-time[1] in which subjective deadlines are set by the end-users. An interesting aspect of reactive applications is that users are usually more perceptive to *variations* in response time rather than absolute performance.

Part of this work investigates the performance of replication techniques in the context of reactive distributed applications. Conventional approaches often exhibit poor performance, and are

[1]In *soft real-time* applications, the violation of deadlines decreases the value of the application. In *hard real-time* applications, the violation of a single deadline compromises the correctness of the application.

subject to long blackout periods, during which the application "freezes," in the case of failures. For various classes of applications, including reactive applications, such blackout periods, no matter how rare, are just unacceptable (e.g., [DMS98]). Indeed, from the end user's point of view, these blackout periods are so long that they are themselves perceived as failures. This dissertation advocates a clear separation between the management of replicas (membership) and the management of failures (failure detection and reconfiguration), in order to avoid long blackout periods. Although the rationale is about reactive applications, the results presented here are general enough to be useful in other contexts, where a good responsiveness in case of failures is as important as in normal cases.

Process controlled crash, or *crash control*, is a major cause for bad response time in the case of failures. In short, process controlled crash is the ability given in some fault-tolerant systems to force processes to crash. This technique is principally used to convert a failure detection mechanism that makes mistakes into one that never does. The principle is simple: whenever a process is suspected to have crashed, it is considered as crashed. It is then killed, "just to make sure!"

The problem with this approach is that it heavily relies on the fact that the underlying failure detection scheme hardly makes any mistake. In order to keep suspicions rare, the failure detection must be extremely conservative, and suspect a process only when it is almost certainly sure that this process has crashed. The caveat is that a conservative failure detection mechanism takes a long time to detect actual failures, thus causing long periods of inactivity. Many current implementations of replicated services rely on the ability to kill processes, hence their poor performance in the occurrence of failures.

1.1.3 Semi-Passive Replication

For both active and passive replication, the responsiveness of the replicated service in case of failure is largely dependent on the replication algorithm. Indeed, as detailed in Chapter 3, a slow reaction to failures is often the result of a tradeoff between a prompt reaction to failures and a fair stability of the system. This tradeoff can nevertheless be lifted by avoiding algorithms that rely on some form of process controlled crash, such as a membership service. While there exist algorithms for active replication that do not require any form of process controlled crash (e.g., [CT96]), this is not the case for passive replication.

Semi-passive replication [DSS98, DS00] is a variant of passive replication that is defined to address this issue. As shown in Chapter 3, the major difference between passive replication and semi-passive replication is that it is possible to devise a semi-passive replication algorithm that does not require any form of process controlled crash. At the same time, semi-passive replication retains the major characteristics of passive replication: a reduced usage of processing power, and the possibility of non-deterministic processing.

In addition, the interactions between client and server are identical in active and semi-passive replication (the client sends to all replicas and waits until it receives the first reply). This is not the case for passive replication, where clients must be able to detect that a primary has crashed and then reissue its request to the new primary.

1.2 Total Order Multicast

Total Order Multicast is an important problem in the field of distributed systems (a special case of this problem is commonly known as *Atomic Broadcast*[2]). In short, Total Order Multicast is a multicast primitive with enhanced guarantees. It ensures that (1) processes deliver messages in the same relative order, and (2) every message is either delivered by all correct processes or by none of them. Atomic Broadcast is defined similarly, but with the additional requirement that messages are sent to all processes in the system.

Total Order Multicast plays for instance a central role when implementing active replication [Sch93, Sch90]. It has also other applications such as clock synchronization [RVC93], computer supported cooperative writing [AS98], distributed shared memory, or distributed locking [Lam78]. More recently, it has also been shown that an adequate use of Atomic Broadcast can significantly improve the performance of replicated databases [AAEAS97, PGS98, KPAS99].

1.2.1 Classification and Survey

In the literature, there exists an incredibly large number of algorithms to solve the problem of Total Order Multicast (or Atomic Broadcast). However, those algorithms are often not equivalent as they not only differ in their mechanism and performance, but also: in the actual properties they guarantee; in the assumptions they make; in the flexibility they provide; or in the way they respond to failures.

In order to better understand and compare Total Order Multicast algorithms, it is necessary to better understand the similarities and dissimilarities between those algorithms. This approach requires to define a classification system, thus grouping algorithms with similar characteristics. In this dissertation, we define such a classification system and use it as a base to survey and compare more than fifty existing algorithms [DSU00].

1.2.2 Metrics and Performance Tradeoffs

Performance is an important factor when choosing an algorithm for a particular application context. One then has to answer the seemingly simple question: "Which algorithm is the most efficient?" The problem considered here is Total Order Multicast for which, as in many cases, there is no clearcut answer to the question. The actual performance of algorithms is essentially a matter of tradeoffs. The identification of these tradeoffs requires a deep understanding and a careful evaluation of the existing solutions to the problem. This also requires a common and unbiased evaluation system.

Distributed algorithms lack the necessary metrics to evaluate and compare the expected performance of group communication algorithms. In the context of sequential algorithms, the need for complexity metrics has been identified and remedied for early. Parallel algorithms also have their tools, such as the various PRAM models in which algorithms can be analyzed and compared. Distributed algorithms are however evaluated with simplistic complexity metrics that fail to pin-

[2]The difference between *broadcast* and *multicast* is explained in Section 4.2.3.

point important tradeoffs related, for instance, to resource utilization. These metrics are useful when one considers large distributed computational algorithms that generate a large number of messages and last a long time. They however reach their limits and lack precision when faced with simpler algorithms such as group communication primitives.

We present two metrics for distributed algorithms that yield more precise information than the existing ones [UDS00a]. Based on these metrics, we analyze and compare the different classes of Total Order Multicast algorithms. The results thus obtained are more relevant and far more detailed than those obtained through more conventional metrics (i.e., time and message complexity).

1.3 Thesis

1.3.1 Assumptions

Observation 1
Some classes of distributed applications require fast reaction to failures, no matter how rarely they may occur (e.g., [DMS98]).

Assumption 1
In fault-tolerant systems, failures must be kept transparent.

Assumption 2
Failures occur independently and are extremely rare.

1.3.2 Claims

Claim 1
Process controlled crash can be avoided in the context of the passive replication technique.

Claim 2
The indiscriminate use of process controlled crash is a major cause of poor performance when failures occur in fault-tolerant group communication systems.

1.3.3 Contribution

There are three main contributions in this dissertation. In the context of replication techniques, there is the definition and the algorithm for semi-passive replication. There are two contributions in the context of Total Order Multicast. The first one is the definition of a classification system for Total Order Multicast and the survey of existing algorithms. The second one is the evaluation and the comparison of those algorithms.

Semi-passive replication Semi-passive replication is a replication technique that retains the major characteristics of passive replication, but that can be implemented without relying on any form of process controlled crash.

Total Order Multicast algorithms: classification and survey Total Order Multicast algorithms are classified according to the mechanism used to generate a total order, thus defining five general classes of algorithms. We then survey more than fifty algorithms that are classified and compared.

Total Order Multicast algorithms: performance tradeoffs The illustration of the performance tradeoffs related to the problem of Total Order Multicast provides a good indication for choosing one algorithm for specific applications contexts. The performance analysis is based on two metrics for distributed algorithms that we have developed.

1.4 Organization

The rest of this dissertation is structured as follows.

- Chapter 2 presents general concepts and definitions that are used throughout this dissertation.
- Chapter 3 introduces semi-passive replication and gives an algorithm for asynchronous systems that does not require any form of process controlled crash.
- Chapter 4 presents a classification system for Total Order Multicast algorithms, and surveys the numerous existing algorithms that solve this problem.
- Chapter 5 defines a set of metrics to predict the performance of distributed algorithms.
- In Chapter 6, the different classes of Total Order Multicast algorithms are compared using the metrics defined in Chapter 5. To this purpose, we define a total of eight algorithms to represent each different class of algorithms.
- Finally, Chapter 7 recalls the main contributions of this work, and discusses some future research directions.

Chapter 2

System Models and Definitions

The usual approach of science of constructing a mathematical model cannot answer the questions of why there should be a universe for the model to describe. Why does the universe go to all the bother of existing?

— **Stephen Hawking** (b.1942)

A system model is a simplification of reality in which problems can be reasoned about. A model can be *general* if it leaves out most irrelevant details, *tractable* if it simplifies the reasoning on actual problems, and *accurate* (or realistic) if it closely matches reality. There is no such thing as a good model: a model is *adequate* to a given problem and environment. In this dissertation, we usually consider the asynchronous system extended with failure detectors, because of its generality and the fact that it is well-defined.

In Section 2.1, we recall the important system models that have been considered in the literature. In Section 2.2, we describe the notion of oracle as an extension to strengthen the system model. We consider three type of oracles: physical clocks, failure detectors, and coin flips. Section 2.3 presents agreement problems, with an emphasis on problems related to group communication. Section 2.4 discusses process controlled crash and some of its implications. Finally, Section 2.5 presents group membership and how it relates to process controlled crash.

2.1 System Models

Distributed systems are modeled as a set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ that interact by exchanging messages through communication channels. There exist a quantity of models that more or less restrict the behavior of these systems components. When one describes a system model, the important characteristics to consider are the synchrony and the failure mode.

2.1.1 Synchrony

The synchrony of a model is related to the timing assumptions that are made on the behavior of processes and communication channels. More specifically, one usually considers two major parameters. The first parameter is the *relative speed of processes*, which is given by the speed ratio of the slowest process with respect to the fastest process in the system. The second parameter is the *communication delay*, which is given by the time elapsed between the emission and the reception of messages. The synchrony of the system is defined by considering various bounds on these two parameters. For each parameter one usually considers the following levels of synchrony.

1. There is a known upper bound which always holds.
2. There is an unknown upper bound which always holds.
3. There is a known upper bound which eventually holds forever.
4. There is an unknown upper bound which eventually holds forever.[1]
5. There is no bound on the value of the parameter.

A system in which both parameters are assumed to satisfy Item 1 is called a *synchronous system*. At the other extreme, a system in which relative process speed and communication delays are unbounded is called an *asynchronous system*. Between those two extremes lie the definitions of various partially synchronous system models [DDS87, DLS88].

2.1.2 Failure Modes

When one considers the problem of failures in a system it is important to specify the kind of such failures. Indeed, there exist various kinds of failures which are not equivalent. The failure mode of a system specifies the kinds of failures that are expected to occur in that system, as well as the conditions under which these failures may or may not occur. The general classes of process failures are the following.

- *Crash failures*. When a process crashes, it ceases functioning forever. This means that it stops performing any activity including sending, transmitting, or receiving any message.
- *Omission failures*. When a process fails by omission, it omits performing some actions such as sending or receiving a message.
- *Timing failures*. A timing failure occurs when a process violates one of the synchrony assumptions. This type of failure is irrelevant in asynchronous systems.
- *Byzantine failures*. Byzantine failures are the most general type of failures. A Byzantine component is allowed any arbitrary behavior. For instance, a faulty process may change the content of messages, duplicate messages, send unsolicited messages, or even maliciously try to break down the whole system.

  In practice, one often considers a particular case of Byzantine failures, called *authenticated* Byzantine failures. Authenticated Byzantine failures allow Byzantine processes to behave

[1]There exist many other possible assumptions, such as: *There is a known upper bound that holds infinitely often for periods of a known duration.*

arbitrarily. However, it is assumed that processes have access to some authentication mechanism (e.g., digital signatures), thus making it possible to detect the forgery of valid messages by Byzantine processes. When mentioning Byzantine failures in the sequel (mostly in Chap.4), we implicitly refer to *authenticated* Byzantine failures.

By contrast, a *correct* process is a process that never expresses any of the faulty behaviors mentioned above. Note that correct/incorrect are predicates over the whole execution: we say that a process that crashes at time t is incorrect already before time t.

Similarly, communication channels can also be subject to crash, omission, timing, and Byzantine failures. For instance, a crashed channel is one that drops every message. Communication failures can for instance cause network partitions. Nevertheless, since communication failures are hardly considered in this dissertation, we do not describe them in further detail.

Peculiarities of Timing Failures

A system is characterized by its "amount of synchrony", and by its failure modes. While the failure mode is normally orthogonal to the synchrony of the system, this is not the case with timing failures which are directly related to the synchrony of the system. Indeed, timing failures are characterized by a violation of the synchrony of the system.

2.2 Oracles

Depending on the synchrony of the system, some distributed problems cannot be solved. Yet, these problems become solvable if the system is extended with an oracle. In short, an oracle is a distributed component that processes can query, and which gives some information that the algorithm can use to guide its choices. In distributed algorithms, at least three different types of oracles are used: (physical) clocks, failure detectors, and coin flips.[2] Since the information provided by these oracles is sufficient to solve problems that are otherwise unsolvable, such oracles augment the power of the system model, and must hence be considered as a part of it.

2.2.1 Physical Clocks

A clock oracle gives information about physical time. Each process has access to its local physical clock and clocks are assumed to give a value that increases monotonically.

The values returned by clocks can also be constrained by further assumptions, such as synchronized clocks. Two clocks are ϵ-synchronized if, at any time, the difference between the values returned by the two clocks is never greater than ϵ. Clocks are not synchronized if ϵ is infinite, and they are perfectly synchronized if $\epsilon = 0$.

Depending on the assumptions, the information returned by the clocks can or cannot be related to real-time. Synchronized clocks are not necessary synchronized with real-time. However, if all local clocks are synchronized with real-time, then they are of course synchronized with each other.

[2]Suggested by Bernadette Charron-Bost.

Note that, with the advent of GPS-based systems, assuming clocks that are perfectly synchronized with real-time is not unrealistic, even in large-scale systems. Indeed, Verı́ssimo, Rodrigues, and Casimiro [VRC97] achieve clock synchronization in the order of a few microseconds, whereas software-based clock synchronization can at best achieve this at a precision several orders of magnitude lower.

2.2.2 Failure Detectors

The notion of failure detectors was formalized by Chandra and Toueg [CT96] in the context of crash failures. Briefly, a failure detector can be seen as a set of distributed modules, one module FD_i attached to every process p_i. Any process p_i can query its failure detector FD_i about the status of other processes (e.g., crashed or not crashed). The information returned by a failure detector FD_i can be incorrect (e.g., a non crashed process can be suspected), and inconsistent (e.g., FD_i can suspect process p_k at time t while, also at time t, FD_j does not suspect p_k). A failure detector is abstractly characterized by a *completeness* and an *accuracy* property. There exist various classes of failure detectors that are defined according to variants of these two basic properties.

Completeness

The completeness of a failure detector is related to its ability to detect crashed processes.

(STRONG COMPLETENESS)
Every process that crashes is eventually forever suspected by *every* correct process.

(WEAK COMPLETENESS)
Every process that crashes is eventually forever suspected by *some* correct process.

Accuracy

The accuracy of a failure detector restricts the incorrect suspicions that the failure detector is allowed to make.

(STRONG ACCURACY)
No process is suspected before it crashes.

(WEAK ACCURACY)
Some correct process is never suspected.

Chandra and Toueg [CT96] also present a variant to the accuracy properties, where the accuracy of failure detectors are restricted only after a certain (unknown) time.

(EVENTUAL STRONG ACCURACY)
There is a time after which *no* correct process is ever 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 2.1: Classes of failure detectors

	Accuracy			
Completeness	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\Diamond\mathcal{P}$	*Eventually Strong* $\Diamond\mathcal{S}$
Weak	$\mathcal{D}$	*Weak* $\mathcal{W}$	$\Diamond\mathcal{D}$	*Eventually Weak* $\Diamond\mathcal{W}$

Classes of Failure Detectors

The eight combinations of completeness and accuracy define as many classes of failure detectors. Each class has a name as illustrated in Table 2.1 [CT96]. An important result presented by Chandra and Toueg [CT96] is that, in a system with crash failures only and reliable channels, strong completeness can be emulated out of weak completeness. This means that any problem that can be solved using a failure detector $\mathcal{P}$ (resp. $\mathcal{S}$, $\Diamond\mathcal{P}$, $\Diamond\mathcal{S}$), can also be solved using a failure detector $\mathcal{D}$ (resp. $\mathcal{W}$, $\Diamond\mathcal{D}$, $\Diamond\mathcal{W}$) instead.

Chandra and Toueg's work on failure detectors is considerably more general than the sole detection of failures, and it can even be argued that this work can actually be extended to encompass the notion of oracle as a whole (Chandra, Hadzilacos, and Toueg [CHT96] give a hint in this direction).

2.2.3 Coin Flip

Another approach to extend the power of a system model consists in introducing the ability to generate random values. This is used by a class of algorithms called randomized algorithms. Those algorithms can solve problems such as consensus with a probability that asymptotically tends to 1 (e.g., [CD89]). It is however important to note that solving a problem for sure and solving a problem with probability 1 are not equivalent. We however do not take this issue further here, as we know of no total order multicast algorithm that is explicitly based on this approach.

2.3 Agreement Problems

Agreement problems constitute a fundamental class of problems in the context of distributed systems. There exist many different agreement problems that share a common pattern: processes have to reach some common decision, the nature of which depends on the problem. This section presents the definition of four fundamental agreement problems: *Reliable Broadcast*, *Atomic Broadcast*, *Consensus*, and *Byzantine Agreement*.

2.3.1 Reliable Broadcast

In short, Reliable Broadcast is a broadcast that guarantees that messages are delivered by either all processes or none. More formally, Reliable Broadcast is defined in terms of the two primitives *R-broadcast*(m) and *R-deliver*(m). When a process executes *R-broadcast*(m), we say that it *broadcasts* message m. Similarly, when a process executes *R-deliver*(m), we say that it *delivers* message m. Reliable Broadcast is defined as follows [HT94]:

(VALIDITY)
If a correct process broadcasts a message m, then it eventually delivers m.

(AGREEMENT)
If a correct process delivers a message m, then all correct processes eventually deliver m.

(INTEGRITY)
For any message m, every correct process delivers m at most once, and only if m was previously broadcast by $sender(m)$.

Hadzilacos and Toueg [HT94] define a hierarchy of Reliable Broadcast specifications, as illustrated on Figure 2.1. The problem of FIFO Reliable Broadcast is defined as a Reliable Broadcast which satisfies the property of FIFO Order:

(FIFO ORDER)
If a process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.

Causal order is defined based on Lamport's [Lam78] "happened before" relation, commonly denoted by "$\longrightarrow$". Let e_i and e_j be three events in a distributed system. The transitive relation $e_i \longrightarrow e_j$ holds if it satisfies the following three conditions: (1) e_i and e_j are two events in the same process, then e_i comes before e_j; (2) e_i is the sending of a message m by one process and e_j is the receipt of m by another process; or (3) There exists a third event e_k such that, $e_i \longrightarrow e_k$ and $e_k \longrightarrow e_j$ (transitivity). Causal Reliable Broadcast is a Reliable Broadcast which satisfies the following causal order property:

(CAUSAL ORDER)
If the broadcast of a message m causally precedes the broadcast of a message m',[3] then no correct process delivers m' unless it has previously delivered m.

2.3.2 Atomic Broadcast

Atomic Broadcast is an extension to Reliable Broadcast. It is defined in terms of two primitives: *A-broadcast*(m) and *A-deliver*(m). In addition to the properties of Reliable Broadcast, it also guarantees that all messages are delivered in the same relative order. This is ensured by the following Total Order property [HT94]:

[3]In other words, *R-broadcast*$(m) \longrightarrow$ *R-broadcast*(m').

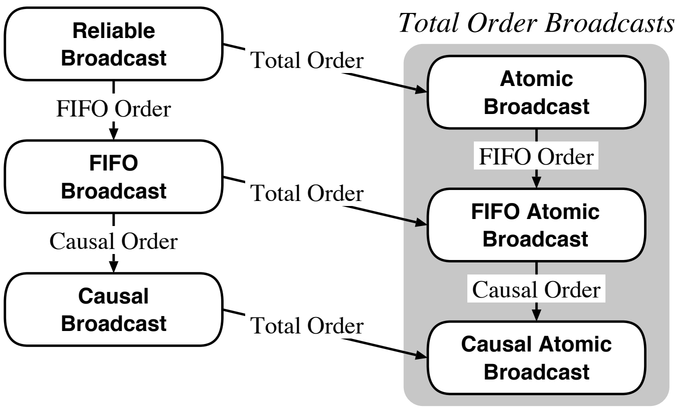


Figure 2.1: Relationship among reliable broadcast primitives

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

As shown by Figure 2.1, the property of Total Order is orthogonal to the other two order properties, namely FIFO order and Causal Order. Thus, the definition of Causal Atomic Broadcast is strictly than that of FIFO Atomic Broadcast, which is in turn stronger than that of "plain" Atomic Broadcast.

2.3.3 Consensus

In the Consensus problem, all correct processes propose a value and the correct processes must decide on one of the proposed values. Consensus is defined in terms of the two primitives *propose*(v) and *deliver*(v). When a process executes *propose*(v), we say that it *proposes* the value v. Similarly, when a process executes *decide*(v), we say that it *decides* on the value v. Consensus is defined as follows [CT96]:

(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.

2.3.4 Byzantine Agreement

The problem of Byzantine Agreement is also well known as the Byzantine Generals Problem [LSP82]. In this problem, every process has an *a priori* knowledge that a particular process s

is supposed to broadcast a single message m. Informally, Byzantine Agreement requires that all correct processes deliver the same message which must be m if the sender is correct. The problem is specified as follows [HT94]:

(TERMINATION)
Every correct process eventually delivers exactly one message.

(VALIDITY)
If s is correct and broadcasts a message m, then it eventually delivers m.

(AGREEMENT)
If a correct process delivers a message m, then all correct processes eventually deliver m.

(INTEGRITY)
If a correct process delivers a message m, then m was sent by s.

As the name indicates, Byzantine Agreement has mostly been studied in relation with Byzantine failures. A variant of Byzantine Agreement, called *Terminating Reliable Broadcast* is sometimes studied in a context limited to crash failures.

2.3.5 Solving Agreement Problems in Asynchronous Systems

There are at least four fundamental theoretical results that are directly relevant to the problem of Atomic Broadcast and Consensus. First, Atomic Broadcast and Consensus are equivalent problems [DDS87, CT96], that is, if there exists an algorithm that solves one problem, then it can be transformed to solve the other problem. Second, there is no (deterministic) solution to the problem of Consensus in asynchronous systems if just a single process can crash [FLP85]. Nevertheless, Consensus can be solved in asynchronous systems extended with failure detectors [CT96], and in some partially synchronous systems [DDS87, DLS88]. Finally, the weakest failure detector to solve Consensus in an asynchronous system is $\Diamond\mathcal{W}$ [CHT96].

2.4 Process Controlled Crash

Process controlled crash is the ability given to processes to kill other processes or to commit suicide. In other words, this is the ability to artificially force the crash of a process. Allowing process controlled crash in a system model increases its power. Indeed, this makes it possible to transform severe failures (e.g., omission, Byzantine) into less severe failures (e.g., crash), and to emulate an “almost perfect” failure detector. However, this power does not come without a price, and process controlled crash should be avoided whenever possible.

2.4.1 Applications of Process Controlled Crash

Process controlled crash has two main applications in a fault-tolerant distributed system. The first application consists in transforming severe failures into crash failures. The second application is to emulate a perfect failure detector in an asynchronous system. These two applications of process controlled crash are powerful, but they come at a very high price, as we discuss at the end of this section.

Automatic Transformation of Failures

Neiger and Toueg [NT90] present a technique to automatically transform some types of failures into less severe ones. In short, this technique is based on the idea that the behavior of processes is monitored. Whenever a process begins behaving incorrectly (e.g., omission, Byzantine), it is killed.[4]

This technique allows to transform algorithms that are tolerant to benign failures (crash or omission) into algorithms that can tolerate stronger types of failures. However, it can only be applied in systems with reliable channels (process omission failures only). This precludes the use of such techniques in systems with lossy channels, or subject to partitions. Indeed, in such contexts, processes might easily be killing each other until not a single one is left alive in the system.

Emulation of an Almost Perfect Failure Detector

In practical systems, perfect failure detectors ($\mathcal{P}$) are extremely difficult to implement because of the difficulty to distinguish crashed processes from very slow ones. Chandra and Toueg [CT96] show that $\mathcal{P}$ can be implemented with timeouts in a synchronous system. However, practical systems are generally quasi-asynchronous.[5]

A perfect failure detector ($\mathcal{P}$) satisfies both strong completeness and strong accuracy. Even in asynchronous systems, it is easy to implement a failure detector that satisfies weak completeness (e.g., [SDS99, CTA00]), and hence strong completeness [CT96] (see Sect. 2.2.2). However, the difficulty is to satisfy strong accuracy (see Sect. 2.2.2). Strong accuracy requires that no process is suspected before it crashes. Process controlled crash makes it possible to emulate the slightly weaker accuracy property that follows:

(QUASI-STRONG ACCURACY)
No correct process is ever suspected by any correct process.

Given a failure detector $\mathcal{X}$ that satisfies strong completeness and any form of accuracy, it is possible to use process controlled crash to emulate a category that we call $\mathcal{AP}$ (almost perfect). The principle is simple: whenever $\mathcal{X}$ suspects a process p, then p is killed (forced to crash). As a result, $\mathcal{AP}$ never makes a mistake *a posteriori*, and thus satisfies the above "quasi-strong accuracy" property. It is interesting to note that $\mathcal{AP}$ is stronger[6] than $\mathcal{S}$ and $\Diamond\mathcal{P}$, but weaker than $\mathcal{P}$.

[4]The actual technique is more complicated than that, but this explanation gives the basic idea.

[5]Practitioners often rely on the assumption that "most messages are *likely* to reach their destination within a known delay δ" [CMA97, CF99]. The existence of such a *finite* bound means that the probabilistic behavior of the network is *known* and *stable*.

[6]informally, a failure detector $\mathcal{D}$ is *stronger* than a failure detector $\mathcal{D}'$ if $\mathcal{D}$ provides more information about failures than $\mathcal{D}'$ does [CT96].

Cost of a Free Lunch

Process controlled crash is often used in practice to solve agreement problems such as Atomic Broadcast and Consensus. It is extremely powerful, but there is a dear price to pay for this. Indeed, process controlled crash reduces the effective fault-tolerance of the system.

To understand this price, it is first necessary to distinguish between two types of crash failures: genuine and provoked failures. *Genuine failures* are failures that naturally occur in the system, without any intervention from a process. Conversely, *provoked failures* are failures that are deliberately caused by some process (murder or suicide), that is, process controlled crash.

A fault-tolerant algorithm can only tolerate the crash of a bounded number of processes.[7] In a system with process controlled crash, this limit not only includes genuine failures, but also provoked failures. This means that each provoked failure actually *decreases* the number of genuine failures that can be tolerated, in other words, it reduces the actual fault-tolerance of the system.

It is important to stress two points. First, in systems with process controlled crash, each occurrence of a suspicion (if it leads to a provoked failure) reduces the number of genuine failures that the system can tolerate. Second, except in very tightly coupled systems, the number of suspicions is generally unpredictable, because message delays are themselves highly unpredictable (e.g., due to unexpected high load). As a result, requiring an upper bound on the number of suspicions makes the system less robust as it may be unable to survive long periods of communication instability.

2.5 Group Membership Service

A group membership service is a distributed service that is responsible for managing groups of processes. It often used as a basic building block for implementing complex group communication systems (e.g., Isis [BVR93], Totem [MMSA$^+$96], Transis [DM96, ADKM92, Mal94], Phoenix [MFSW95, Mal96]), as it allows to keep track of the membership of each process group.

In short, a group membership service is based on the notion of *view*. The view of a group is a list of processes that belong to that group. The membership service must maintain an agreement about the composition of process groups. Thus, it must report changes in this composition to all members, by issuing *view change* notifications.

2.5.1 Management of Process Groups

The main role of a group membership service is obviously to manage the composition of process groups. In a group membership service, processes are managed according to three basic operations. Processes can *join* or *leave* a group during the execution of the system, and processes that are suspected to have crashed are automatically *excluded* from the group. The composition of process groups can thus change dynamically, and processes are informed of these changes when they receive the view change notifications.

[7]The recovery of processes and the dynamic join of new processes are discussed in Section 2.5.

Join & Leave

A process p is allowed to dynamically *join* a process group G by issuing some “join request” message to the membership service. This triggers a view change protocol between the members of the group G, and eventually leads them to accepting a new view that includes p. The protocol normally requires that p be synchronized with the members of G, which is done through a *state transfer*.[8]

When a process p needs to *leave* a process group, it issues some “leave request” message to the membership service. This triggers a view change protocol, such that p does not belong to the new view. Unlike the join operation, a state transfer is normally not necessary during this operation.

Exclusion

When the crash of a process has been detected, this process is removed (or excluded) from any groups it belongs to, thus preventing the system from being clogged by useless crashed processes. Unlike with join and leave operations, the initiative of the view change falls into the hands of the membership service. If a process p is suspected to have crashed, then a new view is proposed from which p is excluded.

However, if p is incorrectly suspected and gets excluded anyway, then p is supposed to commit suicide. What happens in practice is that process p tries to join back the group with a new identity. This means that it will obtain a new state from one of the members of the group, and hence “forget” what happened before it was excluded. This is nothing but a form of process controlled crash, with all its disadvantages.

2.6 Algorithm Notation

2.6.1 Processes and Messages

Table 2.2 to 2.4 present some important aspects of the notation used throughout this dissertation. This section only presents the notation that are fairly general. Some chapters introduce some additional notation specific to their context.

Table 2.2 introduces the notation used to designate the system as a whole. This notation does not depend on the interaction model. Note that the set Π of all processes is static while the subset $\pi(t)$ is dynamic (can vary over time). The purpose of this notation is to simplify the explanations. Although the notion of a dynamic group is an interesting problem in itself, we have decided not to take this issue further in the dissertation.

Table 2.3 extends the notation to the client-server interaction model. This notation is mostly used in Chapter 3 as it is adapted to the description of replicated servers. Note that Π_C is the set of all *potential* clients.

Table 2.4 presents some notation related to messages. The set of messages includes all *application* messages.

[8]A state transfer is an operation in which a process transfers a copy of its state to another process. A detailed discussion on state transfer is however beyond the scope of this dissertation.

Table 2.2: System

Π	set of all processes in the system (static group).
$\pi(t)$	set of all processes in the current group (dynamic group, $\forall t : \pi(t) \subseteq \Pi$).

Table 2.3: Client-server interactions

Π_C	set of client processes.
Π_S	set of server replicas.

Table 2.4: Messages

$\mathcal{M}$	set of all *application* messages sent in the system.
$sender(m)$	sender of message m.
$Dest(m)$	set of destination processes for message m.

Table 2.5: Multicasting

Π_{sender}	set of all sending processes in the system. $\Pi_{sender} \overset{\text{def}}{=} \bigcup_{m \in \mathcal{M}} sender(m)$
Π_{dest}	set of all destination processes in the system. $\Pi_{dest} \overset{\text{def}}{=} \bigcup_{m \in \mathcal{M}} Dest(m)$

Finally, Table 2.5 refines the notation for the context of multicast algorithms. This notation is essentially used in Chapter 4 to describe total order broadcast algorithms.

2.6.2 Sequences

A sequence is defined as a finite ordered list of elements. With a few minor exceptions, the notation defined here is borrowed from Gries and Schneider [GS93].

A sequence of elements a, b, and c is denoted by the tuple $\langle a, b, c\rangle$. The symbol ϵ denotes the empty sequence. The length of a sequence seq is the number of elements in seq and is denoted $\#seq$. For instance,

$$\#(\langle a, b, c\rangle) = 3 \text{ and } \#\epsilon = 0$$

Elements can be added either at the beginning or at the end of a sequence. Adding an element e at the beginning of a sequence seq is called prepending[9] and is denoted by $e \lhd seq$. Similarly, adding an element e at the end of a sequence seq is called appending and is denoted by $seq \rhd e$.

We define the operator [ ] for accessing a single element of the sequence. Given a sequence seq, $seq[i]$ returns the i^{th} element of seq. The element $seq[1]$ is then the first element of the sequence, and is also denoted as $head.seq$. The tail of a non-empty sequence seq is the sequence that results from removing the first element of seq. Thus, we have

$$seq = head.seq \lhd tail.seq$$

[9]The term "prepend" was proposed by Gries and Schneider [GS93] as there was no English word for adding an element at the beginning of a sequence. The word "prepend" is derived from the word "prependant" which means "hanging down in front" (Oxford English Dictionary).

For convenience, we also define the following additional operations on sequences. First, given an element e and a sequence seq, e is a member of seq (denoted $e \in seq$) if e is a member of the set composed of all elements of seq. Second, given a sequence seq and a set of elements S, the exclusion $seq - S$ is the sequence that results from removing from seq all elements that appear in S. Third, given two sequences seq_1 and seq_2, the sequence difference $seq_1 \setminus seq_2$ is the resulting sequence after removing from seq_1 all elements that appear in seq_2.

Chapter 3

Semi-Passive Replication

Deal with the faults of others as gently as with your own.

— Chinese proverb

Replicating a service in a distributed system requires that each replica of the service keeps a consistent state, which is ensured by a specific replication protocol [GS97b]. There exist two major classes of replication techniques to ensure this consistency: *active* and *passive* replication. Both replication techniques are useful since they have complementary qualities.

With active replication [Sch93], each request is processed by all replicas. This technique ensures a fast reaction to failures, and sometimes makes it easier to replicate legacy systems. However, active replication uses all available processing resources and requires the processing of requests to be *deterministic*.[1] This last point is a very strong limitation since, in a program, there exist many potential sources for non-determinism [Pol94]. For instance, multi-threading typically introduces non-determinism.

With passive replication (also called *primary-backup*) [BMST93, GS97b], only one replica (primary) processes the request, and sends update messages to the other replicas (backups). This technique uses less resources than active replication does, without the requirement for operation determinism. On the other hand, the replicated service usually has a slow reaction to failures. For instance, when the primary crashes, the failure must be detected by the other replicas, and the request may have to be reprocessed by a new primary. This may result in a significantly higher response time for the request being processed. For this reason, active replication is often considered a better choice for most real-time systems, and passive replication for most other cases [SM96]. The fact that a single request may have to be processed twice consecutively is inherent to passive replication (time redundancy). However, we show in this chapter that wasting a long time before the failure of a primary gets noticed is not a necessity.

In most computer systems, the implementation of passive replication is based on a synchronous model, or relies on some dedicated hardware device [Pow91, EAP99, AD76, ZJ98, BMST93].

[1]Determinism means that the result of an operation depends only on the initial state of a replica and the sequence of operations it has already performed.

However, we consider here the context of asynchronous systems in which the detection of failures is not certain. In such a system, all implementations of passive replication that we know of are based on a group membership service and must exclude the primary whenever it is suspected to have crashed [BVR93, VRBC93, MMSA$^+$95]. Conversely, there exist implementations of active replication that neither require a group membership service nor need to kill suspected processes (e.g., based on the Atomic Broadcast algorithm proposed by Chandra and Toueg [CT96]).

An algorithm that requires a group membership service for the detection of failures is often prone to long reconfiguration delays in the case of a crash. These delays are caused by the need for conservative failure detection mechanisms. Indeed, incorrect failure suspicions must be extremely rare in order to avoid the excessive price of reconfiguration, and to avoid making the system too unstable. Conversely, an algorithm that is not based on a group membership service can afford more incorrect suspicions, as they induce a significantly lower cost in this context. Hence, such an algorithm can take advantage of an aggressive failure detection mechanism, and thus have a significantly better responsiveness in the case of a crash.

The main contribution of this chapter is to present a replication technique (semi-passive replication) that retains the principal characteristics of passive replication while not requiring a group membership service. We also give a general definition for replication techniques, and prove the correctness of our semi-passive replication algorithm with respect to this definition. The semi-passive algorithm is based on a variant of the Consensus problem called Lazy Consensus, for which we also give a specification, an algorithm, and proofs of correctness.

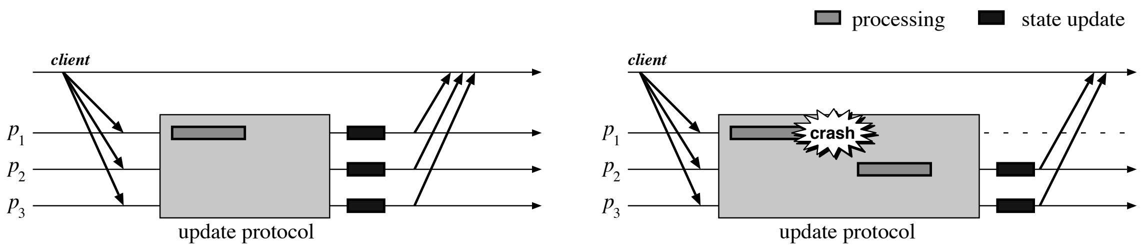
The rest of the chapter is structured as follows. Section 3.1 gives a brief informal overview of semi-passive replication. Section 3.2 proposes a formal definition of replication. In Section 3.3, we give an algorithm for semi-passive replication, and present the problem of Lazy Consensus on which it is based. In Section 3.4 we present an algorithm for the Lazy Consensus problem designed for asynchronous systems augmented with a failure detector. Section 3.5 discusses how semi-passive replication relates to other replication techniques, and discusses the roles of a group membership service.

3.1 Overview of Semi-Passive Replication

The passive replication technique is quite useful in practice, since it requires less processing power than active replication and makes no assumption on the determinism of processing a request. Semi-passive replication can be seen as a variant of passive replication, as it retains most major characteristics (e.g., allows non-deterministic processing). However, the selection of the primary is based on the rotating coordinator paradigm [CT96]—a mechanism simpler than a group membership service.

Informally, semi-passive replication works the following way. The client sends its request to all replicas p_1, p_2, p_3 (see Fig. 3.1(a)). The servers know that p_1 is the first primary, so p_1 handles the requests and updates the other servers.

If p_1 crashes and is not able to complete its job as the primary, or if p_1 does not crash but is incorrectly suspected of having crashed, then p_2 takes over as the new primary. The details of



(a) No crash.

(b) Crash of the coordinator.

Figure 3.1: Semi-passive replication (conceptual representation: the *update protocol* actually hides several messages)

how this works are explained later in the chapter. Figure 3.1(b) illustrates a scenario in which p_1 crashes after handling the request, but before sending its update message. After the crash of p_1, p_2 becomes the new primary.

These examples do not show which process is the primary for the next client requests, nor what happens if client requests are received concurrently. These issues are explained in detail in Section 3.3. However, the important point in this solution is that no process is ever excluded from the group of servers (as in a solution based on a membership service). In other words, in case of false suspicion, there is no join (and state transfer) that needs to be executed by the falsely suspected process. This significantly reduces the cost related to an incorrect failure suspicion, i.e., the cost related to the aggressive timeout option mentioned before.

3.2 Properties of Replication Techniques

Although there exist specifications for passive replication [BMST93, Bud93], and active replication [Sch93], those specifications are too specific and fail to describe properties that are common to *all* replication techniques. In this section, we give four properties that any replication technique should satisfy.

3.2.1 Notation

We define a replication technique in the context of the client-server model. We consider two types of processes: (1) clients and (2) replicas of a server. The set of all clients in the system is denoted by Π_C, and the set of server replicas is denoted by Π_S.[2] We also denote the number of server processes by $n = |\Pi_S|$.

The clients issue requests to a (replicated) server, and receive replies from this server. We consider that a server is replicated when it is composed of more than one process (called replicas hereafter). The replicas update their respective state according to the requests issued by the clients.

[2]Note that Π_C and Π_S may overlap.

We define three events in the system. The event $send(req)$ corresponds to the emission of the request req by a client. The event $update(req)$ corresponds to the modification of the state of a replica, according to the request req. Finally, the event $receive(resp_{req})$ corresponds to the reception by a client of the response to request req (message $resp_{req}$).

3.2.2 Properties

In this section, we give a formal definition for replication techniques in general. Any replication technique should hence satisfy the following properties.

(TERMINATION)
: If a correct client $c \in \Pi_C$ sends a request, it eventually receives a reply.

(AGREED ORDER)
: If there is an event $update(req)$ such that a replica executes $update(req)$ as its i^{th} update event, then all replicas that execute the i^{th} update event also execute $update(req)$ as their i^{th} event.

(UPDATE INTEGRITY)
: For any request req, every replica executes $update(req)$ at most once, and only if $send(req)$ was previously executed by a client.

(RESPONSE INTEGRITY)
: For any event $receive(resp_{req})$ executed by a client, the event $update(req)$ is executed by some correct replica.

3.3 Semi-Passive Replication

We begin this section by giving a general overview of semi-passive replication. First, we introduce the problem of *Lazy Consensus*, on which our semi-passive replication algorithm is based. We then present our algorithm for semi-passive replication, expressed as a sequence of Lazy consensus problems. Following this, we give the proof of correctness for our semi-passive replication algorithm. Finally, we give and prove additional properties that are specific to semi-passive replication.

3.3.1 Basic Idea: Consensus on "update" values

As mentioned in Section 3.1, in the semi-passive replication technique, the requests are handled by the primary. After the processing of each request, the primary sends an *update* message to the backups.

Our solution is based on a sequence of Lazy Consensus problems, in which every consensus problem decides on the *content of the update message*. This means that the initial value of every consensus problem is an *update value*, generated when handling the request. The cost related to getting the initial value is high as it requires the processing of the request. So, we want to avoid a situation in which each server processes such a request, i.e., has an initial value for consensus (the semi-passive replication technique could no more be qualified as "passive"). This explains the need for a "laziness" property regarding the Consensus problem.

Expressing semi-passive replication as a sequence of Lazy Consensus problems hides the role of the primary inside the consensus algorithm. A process p takes the role of the primary (i.e., handles client requests) exactly when it proposes its initial value.

3.3.2 Lazy Consensus

The Lazy Consensus problem is a generalization of the Consensus [CT96] problem, which allows processes to delay the computation of their initial value. In the standard Consensus problem each process starts with an initial value. In the Lazy Consensus each process gets its initial value only when necessary.

Problem

We consider the set of processes Π_S. With the standard Consensus [CT96], processes in Π_S begin solving the problem by proposing a value. This is in contradiction with the laziness property that we want to enforce. So, processes begin solving the problem by calling the procedure *LazyConsensus*(giv), where giv is an argument-less function[3] that computes an initial value v (with $v \neq \bot$[4]) and returns it. When a process p calls giv, we say that p *proposes* the value v returned by giv. When a process q executes $decide(v)$, we say that q *decides* the value v. The Lazy Consensus problem is specified in Π_S by 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.

(PROPOSITION INTEGRITY)
Every process proposes a value at most once.

(STRONG LAZINESS)
If two processes p and q propose a value, then p or q is incorrect.

Laziness is the only new property with respect to the standard definition of the Consensus problem. We also propose a weaker definition for the Lazy Consensus problem. Indeed, the strong laziness property can only be ensured in a system in which the detection of failures is certain.

[3] giv stands for *get initial value*.

[4] The symbol $\bot$ (bottom) is a common way to denote the absence of value. This is called either *nil* or *null* in most programming languages.

(WEAK LAZINESS)

If two processes p and q propose a value, then p or q is suspected by some[5] process in Π_S.

With weak laziness, some incorrect suspicions are treated in the same way as an actual crash. This means that, in some cases, an incorrect suspicion may lead two correct processes to propose a value. When this occurs, the actual cost on the system is related to the price paid each time a process proposes a value. In practice, if failure detectors are tuned properly, the probability of incorrectly suspecting one specific process should remain low.

In the sequel, we present an algorithm for semi-passive replication that is based on Lazy Consensus, and makes use of its laziness property. Solving Lazy Consensus is discussed in Section 3.4.

3.3.3 Semi-Passive Replication Algorithm

We now express semi-passive replication using Lazy Consensus. The algorithm relies on the laziness property of the Lazy Consensus, to ensure that a request is handled by only one replica when no failure occurs.

Notation and System Model

In order to express our semi-passive replication algorithm, we introduce the following notation.

- req: request message sent by a client (denoted by $sender(req)$).
- upd_{req}: update message generated after handling request req.
- $resp_{req}$: response message to the client $sender(req)$, generated after handling request req.
- $state_s$: the state of process s.
- $handle : (req, state_s) \longmapsto (upd_{req}, resp_{req})$
  Processing of request req by the server s in $state_s$. The result is an update message upd_{req} and the corresponding response message $resp_{req}$.
- $update : (upd_{req}, state_{s'}) \longmapsto state'_{s'}$
  Returns a new state $state'_{s'}$, defined by the application of the update message upd_{req} to the state $state_{s'}$. This corresponds to the event $update(req)$ mentioned in Section 3.2, where s' is the server that executes *update*.

We express the semi-passive replication algorithm as two tasks that can execute concurrently (Alg. 3.1). A task with a clause **when** is enabled when the condition is true. We assume that (1) any task that is enabled is eventually executed, and (2) there are no multiple concurrent executions of the same task (Task 1 or Task 2 in Algorithm 3.1). We also make the following assumptions regarding the system:

- processes fail by crashing only (no Byzantine failures);

[5] As a matter of fact, the Lazy Consensus algorithm presented in this dissertation satisfies a stronger property: two processes propose a value only if one is suspected by a *majority* of processes in Π_S (Lemma 21, p. 39).

- communication channels are reliable (no message loss if sender and destination are correct);
- crashed processes never recover.

We make these assumptions in order to simplify the description of the algorithms. Indeed, based on the literature, the algorithms can easily be extended to lossy channels and network partitions [ACT97, ACT99], and to handle process recovery [ACT98, OGS97, HMR98]. However, this would obscure the key idea of semi-passive replication by introducing unnecessary complexity.

Algorithm

We now give the full algorithm, which expresses the semi-passive replication technique as a sequence of Lazy Consensus problems. The algorithm, executed by every server process, is given in Algorithm 3.1.

Every server s manages an integer k (line 5), which identifies the current instance of the Lazy Consensus problem. Every server process also handles the variables $recv$ and $hand$ (lines 2,3):

Algorithm 3.1 Semi-passive replication (code of server s)

```
1:  Initialisation:
2:     $recvQ_s \leftarrow \epsilon$                                  {sequence of received requests, initially empty}
3:     $hand_s \leftarrow \emptyset$                                  {set of handled requests}
4:     $state_s \leftarrow state^0$
5:     $k \leftarrow 0$

6:  function handleRequest()
7:     $req \leftarrow head.recvQ_s$
8:     $(upd_{req}, resp_{req}) \leftarrow handle(req, state_s)$
9:     return $(req, upd_{req}, resp_{req})$

10: when receive($req_c$) from client $c$                             {TASK 1}
11:    if $req_c \notin hand \wedge req_c \notin recvQ_s$ then
12:       $recvQ_s \leftarrow recvQ_s \rhd req_c$

13: when $\#recvQ_s > 0$                                              {TASK 2}
14:    $k \leftarrow k + 1$
15:    LazyConsensus($k$,handleRequest)                               {Solve the $k^{th}$ Lazy consensus}
16:    wait until $(req, upd_{req}, resp_{req}) \leftarrow decided$
17:    send $(resp_{req})$ to $sender(req)$                            {Send response to client}
18:    $state_s \leftarrow update(upd_{req}, state_s)$                   {Update the state}
19:    $recvQ_s \leftarrow recvQ_s - \{req\}$
20:    $hand_s \leftarrow hand_s \cup \{req\}$
```

- $recvQ_s$ is a sequence containing the requests received by a server s, from the clients. Whenever s receives a new request, it is appended to $recvQ_s$ (lines 10,12);
- $hand_s$ is a set which consists of the requests that have been processed. A new Lazy Consensus is started whenever the preceding one terminates, and the receive queue $recvQ_s$ is not

empty (line 13). At the end of the consensus, the request that has been processed (handling and update) is removed from $recvQ_s$ and inserted into $hand_s$ (line 20).

The main part of the algorithm consists in lines 13–20: at line 15 the Lazy Consensus algorithm is started. Then, the server waits for the decision value $(req, upd_{req}, resp_{req})$ of Consensus: req is the request that has been handled; upd_{req} is the update resulting from handling req; and $resp_{req}$ is the response that should be sent to the client.

At line 17 the response $resp_{req}$ is sent to the client. At line 18 the local state of the server s is updated according to the update message upd_{req}. Finally, at line 20 the request that has been handled is inserted into the set $hand$.

The function *handleRequest* The function *handleRequest* returns initial values for each instance of the Lazy consensus problem. The function *handleRequest* (line 6) is called by the Lazy consensus algorithm. In other words, when a process calls *handleRequest*, it acts as a primary in the context of the semi-passive replication technique.

The function *handleRequest* selects a client request that has not been handled yet (line 7), handles the request (line 8), and returns the selected request req, the update message upd_{req} resulting from handling req, as well as the corresponding response message $resp_{req}$ (line 9).

3.3.4 Proof of Correctness

We prove that our algorithm for semi-passive replication satisfies the properties given in Section 3.2. The proof assumes that (1) procedure *LazyConsensus* solves the Lazy Consensus problem according to the specification given in Section 3.3.2 (ignoring the laziness property), and (2) at least one replica is correct. Solving Lazy Consensus is discussed in Section 3.4. In fact, Lazy Consensus solves Consensus, which is enough for the correctness of Replication. A reader not interested in the details of the proofs may skip to Section 3.3.5.

Lemma 1 (Termination) *If a correct client $c \in \Pi_C$ sends a request, it eventually receives a reply.*

PROOF. The proof is by contradiction. Let req_c be a request sent by a correct client c that never receives a reply. As c is correct, all correct replicas in Π_S eventually receive req_c at line 10, and insert req_c into their receive queue $recvQ_s$ at line 11. By the assumption that c never gets a reply, no correct replica decides at line 14 on $(req_c, -,)$: if one correct replica would decide, then by the Agreement and Termination property of Lazy Consensus, all correct replicas would decide on $(req_c, -, -)$. As we assume that there is at least one correct replica then, by the property of the reliable channels, and because c is correct, c would eventually receive a reply. Consequently, req_c is never in $hand$ of any replica, and thus no replica s removes req_c from $recvQ_s$ (Hyp.1.1).

Let t_0 be the earliest time such that the request req_c has been received by every replica that has not crashed. Let $beforeReqC_s$ denote the prefix of sequence $recvQ_s$ that consists of all requests in $recvQ_s$ that have been received before req_c. After t_0, no new request can be inserted in $recvQ_s$ before req_c, and hence none of the sequences $beforeReqC$ can grow.

Let l be the total number of requests that appear before req_c in the receive queue $recvQ_s$ of any replica s that has not crashed:

$$l = \sum_{s \in \Pi_S} \begin{cases} 0 & \text{if } s \text{ has crashed} \\ \#beforeReqC_s & \text{otherwise} \end{cases}$$

After time t_0, the value of l cannot increase since all new requests can only be inserted *after* req_c. Besides, after every decision of the Lazy Consensus at line 16, at least one replica s' removes the request $req_{h_{s'}}$ at the head of $recvQ_{s'}$ (l.7, l.19). The request $req_{h_{s'}}$ is necessarily before req_c in $recvQ_{s'}$, and hence belongs to $beforeReqC_{s'}$. As a result, every decision of the Lazy Consensus leads to decreasing the value of l by at least 1.

Since req_c is never removed from $recvQ_s$ (by Hyp.1.1), Task 2 is always enabled ($\#recvQ_s \geq 1$). So, because of the Termination property of Lazy Consensus, the value of l decreases and eventually reaches 0 (this is easily proved by induction on l).

Let t_1 be the earliest time at which there is no request before req_c in the receive queue $recvQ$ of any replica ($l = 0$). This means that, at time t_1, req_c is at the head of the receive queue of all running replicas, and the next execution of Lazy Consensus can only decide on request req_c (l.7). Therefore, every correct replica s eventually removes req_c from $recvQ_s$, a contradiction with Hyp.1.1. □

Lemma 2 (Agreed order) *If there is an event $update(req)$ such that a replica executes the event $update(req)$ as its i^{th} update event, then all replicas that execute the i^{th} update event also execute $update(req)$ as their i^{th} event.*

PROOF. Let some replica execute $update(req)$ as the i^{th} update, i.e., the replica reaches line 18 of the algorithm with $k = i$ and executes $state \leftarrow update(upd_{req}, state)$. This means that the replica has decided on $(req, upd_{req}, -)$ at line 16. By the Agreement property of Lazy Consensus, every replica that decides for $k = i$, also decides on $(req, upd_{req}, -)$ at line 16. Therefore, every replica that executes line 18 with $k = i$ also executes $state \leftarrow update(upd_{req}, state)$. □

Lemma 3 *If a replica executes $update(req)$, then $send(req)$ was previously executed by a client.*

PROOF. If a replica p executes $update(req)$, then some replica q has selected and processed the request req at line 7 and line 8 respectively. It follows that req was previously received by q, as req belongs to the sequence $recvQ_s$. Therefore, req was sent by some client. □

Lemma 4 *For any request req, every replica executes $update(req)$ at most once.*

PROOF. Whenever a replica executes $update(req)$ (line 18), it has decided on $(req, -, -)$ at line 15, and inserts req into the set of handled requests $hand$ (line 18). By the Agreement property of Lazy Consensus, every replica that decides at line 15 decides also on $(req, -, -)$ and inserts also req into hand at line 18. As a result, no replica can select req again at line 7, and $(req, -, -)$ cannot be the decision of any subsequent Lazy Consensus. □

Lemma 5 (Update integrity) *For any request req, every replica executes $update(req)$ at most once, and only if $send(req)$ was previously executed by a client.*

PROOF. The result follows directly from Lemma 3 and Lemma 4. □

Lemma 6 (Response integrity) *For any event $receive(resp_{req})$ executed by a client, the event $update(req)$ is executed by some correct replica.*

PROOF. If a client receives $resp_{req}$, then $send(resp_{req})$ was previously executed by some replica (line 16). Therefore, this replica has decided $(req, upd_{req}, res_{req})$ at line 15. By the Termination and Agreement properties of Lazy Consensus, every correct replica also decides $(req, upd_{req}, res_{req})$ at line 15, and executes $update(req)$ at line 17. The lemma follows from the assumption that at least one replica is correct. □

Theorem 7 *Algorithm 3.1 solves the replication problem.*

PROOF. Follows directly from Lemma 2 (agreed order), Lemma 5 (update integrity), Lemma 6 (response integrity), and Lemma 1 (termination). □

3.3.5 Additional Properties of the Semi-Passive Replication Algorithm

The proofs of correctness do not rely on the laziness property of the Lazy Consensus. However, without additional properties, the semi-passive replication can hardly be qualified as passive. Indeed, "passive" means that normally only one process processes each request. Therefore, we introduce a property of *parsimony* that expresses this idea. For the same reason that we have given a weak and a strong version of laziness for the Lazy Consensus, we give here a weak and a strong version of the parsimony property. The parsimony property is totally dependent on the laziness property of the Lazy Consensus. As a result, strong parsimony requires strong laziness, and weak parsimony only requires weak laziness.

Lemma 8 (Strong parsimony) *If Lazy Consensus satisfies the strong laziness property, then if two replicas p and q process a request req, then p or q is incorrect.*

PROOF. Processes process a request at line 8, when they propose a value. Therefore, this follows directly from the *strong laziness* property of Lazy Consensus. □

Lemma 9 (Weak parsimony) *If Lazy Consensus satisfies the weak laziness property, then if two replicas p and q process a request req, then p or q is suspected by some process in Π_S.*

PROOF. Follows directly from the *weak laziness* property of Lazy Consensus. □

It is important to stress again that laziness, whether strong or weak, does not influence the correctness of the replication technique. The ability of the Lazy Consensus algorithm to satisfy the laziness property merely influences the *efficiency* of the semi-passive replication.

Lemma 10 (Non-determinism) *The processing of requests does not need to be deterministic.*

PROOF. Follows directly from Algorithm 3.1 and the use of Lazy Consensus to agree on the update upd_{req} and on the reply $resp_{req}$ for a request req. □

3.4 Lazy Consensus

In this section, we give an algorithm that solves the problem of Lazy Consensus defined in Section 3.3.2.[6] The algorithm is based on the asynchronous model augmented with failure detectors [CT96].

We first give a brief description of the system model for solving Lazy Consensus and we give an algorithm that solves the Lazy Consensus problem using the $\Diamond\mathcal{S}$ class of failure detectors. We then present two scenarios of the semi-passive replication algorithm. Finally, we propose an extension to our Lazy Consensus algorithm that improves its efficiency in case of failures.

3.4.1 System Model

In order to solve Lazy Consensus among the server processes Π_S, we consider an asynchronous system augmented with failure detectors [CT96]. Semi-passive replication could easily be expressed in other system models in which Lazy Consensus is solvable. We have chosen the failure detector model for its generality and its conceptual simplicity. We assume here the $\Diamond\mathcal{S}$ failure detector defined on the set of server processes Π_S, which is defined by the following properties (see Sect. 2.2.2):

(STRONG COMPLETENESS)
There is a time after which every process in Π_S that crashes is permanently suspected by all correct processes in Π_S.

(EVENTUAL WEAK ACCURACY)
There is a time after which some correct process in Π_S is never suspected by any correct process in Π_S.

[6]An earlier version of this algorithm was called $\mathcal{DIV}$consensus [DSS98].

3.4.2 Lazy Consensus Algorithm using $\Diamond\mathcal{S}$

The algorithm for Lazy Consensus is adapted from the consensus algorithm using $\Diamond\mathcal{S}$ proposed by Chandra and Toueg [CT96]. This algorithm assumes that a majority of the processes are correct ($f = \lfloor \frac{n}{2} \rfloor$, where f is the maximum number of processes that may crash). The algorithm proceeds in asynchronous rounds, and is based on the rotating coordinator paradigm: in every round another process is coordinator.

Note that different algorithms for Lazy Consensus using $\Diamond\mathcal{S}$ can be adapted from other consensus algorithms based on the rotating coordinator paradigm (e.g., [Sch97, HR97, MR99]).

Solving the Lazy Consensus Problem

Algorithm 3.2 is an algorithm to solve the Lazy Consensus problem. There are actually only little difference between the Algorithm 3.2 and the $\Diamond\mathcal{S}$ consensus algorithm of [CT96]: the lines in which the two algorithms differ are highlighted with an arrow in the margin on Figure 3.2. We do not give a full explanation of the algorithm as the details can be found in [CT96]. However, we explain the parts on which the algorithms differ.

Compared with Chandra and Toueg's algorithm, we have the following three differences.

1. *Laziness (Alg. 3.2, lines 4,18,23–26).* The most important difference is the modifications necessary to satisfy the laziness property.

2. *Dynamic system list (Alg. 3.2, lines 2,5,10,27,33,46).* The ability of the algorithm to reorder the list of processes improves its efficiency in the case of a crash (in the case of a sequence of consensus executions).

3. *Optimization of the first phase (Alg. 3.2, lines 13,17).* The first phase of the first round of the consensus algorithm is not necessary for the correctness of the algorithm [Sch97, DFS99]. We have thus optimized it away.

Laziness

In the context of a solution based on the rotating coordinator paradigm, a process p_i proposes a value only when it is coordinator, and only if p_i is not aware of any previously computed initial value. So, in the absence of failures (and suspicions), only one process calls the function giv (see Sect. 3.3.3). Figure 3.2 illustrates a run of the algorithm in the absence of crash (and incorrect failure suspicions). Only the coordinator of the first round (process p_1) calls the function giv, and the Lazy Consensus problem is solved in one round. If p_1 crashes, two rounds may be needed to solve the problem: process p_1, coordinator of the first round, calls giv and crashes. The surviving processes move to the second round. Process p_2, coordinator of the second round, in turn calls giv.

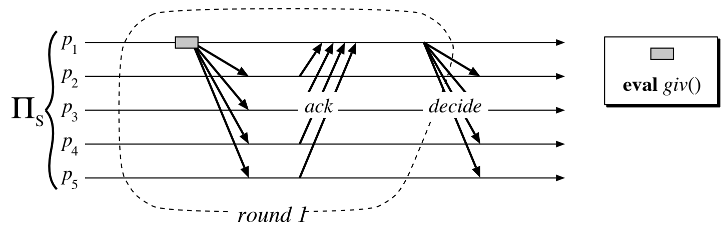


Figure 3.2: Lazy Consensus with $\Diamond\mathcal{S}$ (no crash, no suspicion)

Dynamic System List

In the rotating coordinator paradigm, every instance of the Chandra and Toueg's Consensus algorithm invariably starts the first round by selecting the same process, say p_1, as the coordinator. If p_1 crashes, further executions of the consensus algorithm will always require at least two rounds to complete. This extra cost (two rounds instead of one) is avoided by a very simple idea.

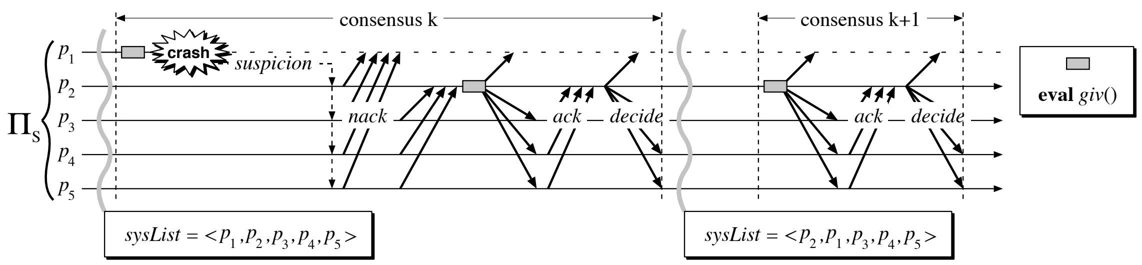


Figure 3.3: Permutations of Π_S and selection of the coordinator

Assume that, for the consensus number k, the processes of Π_S are ordered $[p_1, p_2, p_3, p_4, p_5]$, which defines p_1 as the first coordinator (see Fig. 3.3). If p_1 is suspected (e.g., it has crashed) during consensus k, the processes of Π_S are reordered $[p_2, p_3, p_4, p_5, p_1]$ for the consensus $k+1$, which defines p_2 as the first coordinator. So, despite the crash of p_1 in consensus k, consensus $k+1$ can be solved in one single round.

This reordering, implemented in the context of the Lazy Consensus algorithm, requires no additional message. In the algorithm, processes not only try to reach an agreement on the decision value, but also on the order of the system list. For this purpose, they manage two estimate variables: *estV*$_p$ for the decision value, and *estL*$_p$ for the system list. When a coordinator proposes a value, it also proposes a system list in which it is the first coordinator (see Alg. 3.2, line 5 and 28).

Optimization of the First Phase

Compared with Chandra and Toueg's algorithm, a further difference consists in the optimization of the first phase of the algorithm [Sch97]. In the Lazy Consensus algorithm, all processes start with $\perp$ as their estimate. Consequently, the coordinator of the first phase cannot expect anything but $\perp$ from the other processes. Hence, in the first round, the algorithm skips the first phase and proceeds directly to the second phase in line 18.

Algorithm 3.2 Lazy Consensus (code of process p)

```
    1: Initialisation:
→  2:     $sysList_p \leftarrow \langle p_1, p_2, \ldots, p_n \rangle$

    3: procedure Lazy$Consensus$ ( function $giv : \emptyset \mapsto v$)
→  4:     $estV_p \leftarrow \bot$                                    {$p$'s estimate of the decision value}
→  5:     $estL_p \leftarrow p \lhd (sysList_p - \{p\})$          {$p$'s estimate of the new system list (with $p$ at the head)}
    6:     $state_p \leftarrow undecided$
    7:     $r_p \leftarrow 0$                                        {$r_p$ is $p$'s current round number}
    8:     $ts_p \leftarrow 0$                         {$ts_p$ is the last round in which $p$ updated $estV_p$, initially 0}
    9:     while $state_p = undecided$ do              {rotate through coordinators until decision reached}
→ 10:         $c_p \leftarrow sysList_p\,[(r_p \bmod n) + 1]$                 {$c_p$ is the current coordinator}
   11:         $r_p \leftarrow r_p + 1$
   12:         Phase 1:                               {all processes $p$ send $estV_p$ to the current coordinator}
→ 13:             if $r_p > 1$ then
   14:                 send $(p, r_p, estV_p, estL_p, ts_p)$ to $c_p$
   15:         Phase 2:             {coordinator gathers $\lceil \frac{n+1}{2} \rceil$ estimates and proposes new estimate}
   16:             if $p = c_p$ then
→ 17:                 if $r_p = 1$ then
→ 18:                     $estV_p \leftarrow$ eval $giv()$                             {$p$ proposes a value}
→ 19:                 else
   20:                     wait until [for $\lceil \frac{n+1}{2} \rceil$ processes $q$ : received $(q, r_p, estV_q, estL_q, ts_q)$ from $q$]
   21:                     $msgs_p[r_p] \leftarrow \{(q, r_p, estV_q, estL_q, ts_q) \mid p$ received $(q, r_p, estV_q, estL_q, ts_q)$ from $q\}$
   22:                     $t \leftarrow$ largest $ts_q$ such that $(q, r_p, estV_q, estL_q, ts_q) \in msgs_p[r_p]$
→ 23:                     if $estV_p = \bot$ and $\forall (q, r_p, estV_q, estL_q, ts_q) \in msgs_p[r_p] : estV_q = \bot$ then
→ 24:                         $estV_p \leftarrow$ eval $giv()$                         {$p$ proposes a value}
→ 25:                     else
→ 26:                         $estV_p \leftarrow$ select one $estV_q \neq \bot$ s.t. $(q, r_p, estV_q, estL_q, t) \in msgs_p[r_p]$
→ 27:                         $estL_p \leftarrow estL_q$
→ 28:                 send $(p, r_p, estV_p, estL_p)$ to all
   29:         Phase 3:                   {all processes wait for new estimate proposed by current coordinator}
   30:             wait until [received $(c_p, r_p, estV_{c_p}, estL_{c_p})$ from $c_p$ or $c_p \in \mathcal{D}_p$]   {query failure detector $\mathcal{D}_p$}
   31:             if [received $(c_p, r_p, estV_{c_p}, estL_{c_p})$ from $c_p$] then              {$p$ received $estV_{c_p}$ from $c_p$}
   32:                 $estV_p \leftarrow estV_{c_p}$
→ 33:                 $estL_p \leftarrow estL_{c_p}$
   34:                 $ts_p \leftarrow r_p$
   35:                 send $(p, r_p, ack)$ to $c_p$
   36:             else                                                  {$p$ suspects that $c_p$ crashed}
   37:                 send $(p, r_p, nack)$ to $c_p$
   38:         Phase 4: {the current coordinator waits for $\lceil \frac{n+1}{2} \rceil$ replies. If they indicate that $\lceil \frac{n+1}{2} \rceil$ processes adopted its
                 estimate, the coordinator R-broadcasts a decide message}
   39:             if $p = c_p$ then
   40:                 wait until [for $\lceil \frac{n+1}{2} \rceil$ processes $q$ : received $(q, r_p, ack)$ or $(q, r_p, nack)$]
   41:                 if [for $\lceil \frac{n+1}{2} \rceil$ processes $q$ : received $(q, r_p, ack)$] then
   42:                     R-broadcast $(p, r_p, estV_p, estL_p, decide)$

   43: when R-deliver $(q, r_q, estV_q, estL_q, decide)$     {if $p$ R-delivers a decide message, $p$ decides accordingly}
   44:     if $state_p = undecided$ then
   45:         decide($estV_q$)
→ 46:         $sysList_p \leftarrow estL_q$                       {updates the system list for the next execution}
   47:         $state_p \leftarrow decided$
```

3.4.3 Proof of Correctness

Here, we prove the correctness of Algorithm 3.2. The algorithm solves the weak Lazy Consensus problem using the $\Diamond\mathcal{S}$ failure detector. Lemma 12–15 are adapted from the proofs of Chandra and Toueg [CT96] for the Consensus algorithm with $\Diamond\mathcal{S}$. A reader not interested in the details of the proofs may skip to Section 3.4.4.

Lemma 11 *No correct process remains blocked forever at one of the* wait *statements.*

PROOF. There are three *wait* statements to consider in Algorithm 3.2 (l.20, l.30, l.40). The proof is by contradiction. Let r be the smallest round number in which some correct process blocks forever at one of the *wait* statements.

In Phase 2, we must consider two cases:

1. If r is the first round, then the current coordinator $c = SysList[1]$ does not wait in Phase 2 (l.17), hence it does not block in Phase 2.

2. If $r > 1$ then, all correct processes reach the end of Phase 1 of round r, and they all send a message of the type $(-, r, estV, -, -)$ to the current coordinator $c = sysList[((r - 1) \bmod n) + 1]$ (l.14). Since a majority of the processes are correct, at least $\lceil \frac{n+1}{2} \rceil$ such messages are sent to c and c does not block in Phase 2.

For Phase 3, there are also two cases to consider:

1. c eventually receives $\lceil \frac{n+1}{2} \rceil$ message of the type $(-, r, estV, -, -)$ in Phase 2.

2. c crashes.

In the first case, every correct process eventually receives $(c, r, estV_c, -)$ (l.30). In the second case, since $\mathcal{D}$ satisfies strong completeness, for every correct process p there is a time after which c is permanently suspected by p, that is, $c \in \mathcal{D}_p$. Thus in either case, no correct process blocks at the second *wait* statement (Phase 3, l.30). So every correct process sends a message of the type $(-, r, ack)$ or $(-, r, nack)$ to c in Phase 3 (resp. l.35, l.37). Since there are at least $\lceil \frac{n+1}{2} \rceil$ correct processes, c cannot block at the *wait* statement of Phase 4 (l.40). This shows that all correct processes complete round r—a contradiction that completes the proof of the lemma. □

Lemma 12 (Termination) *Every correct process eventually decides some value.*

PROOF. There are two possible cases:

1. **Some correct process decides.** If some correct process decides, then it must have R-delivered some message of type $(-, -, -, -, decide)$ (l.43)). By the agreement property of Reliable Broadcast, all correct processes eventually R-deliver this message and decide.

2. **No correct process decides.** Since $\mathcal{D}$ satisfies eventual weak accuracy, there is a correct process q and a time t such that no correct process suspects q after time t. Let $t' \geq t$ be a time such that all faulty processes crash. Note that after time t' no process suspects q. From this and Lemma 11, because no correct process decides there must be a round r such that:

   (a) All correct processes reach round r after time t' (when no process suspects q).

   (b) q is the coordinator of round r (i.e., $q = sysList[((r-1) \bmod n)+1]$).

   Since q is correct, then it eventually sends a message to all processes at the end of Phase 2 (l.28):

   - If round r is the first round, then q does not wait for any message, and sends $(q, r, estV_q, -)$ to all processes at the end of in Phase 2.
   - For round $r > 1$, then all correct processes send their estimates to q (l.14). In Phase 2, q receives $\lceil\frac{n+1}{2}\rceil$ such estimates, and sends $(q, r, estV_q, -)$ to all processes.

   In Phase 3, since q is not suspected by any correct process after time t, every correct process waits for q's estimate (l.30), eventually receives it, and replies with an *ack* to q (l.35). Furthermore, no process sends a *nack* to q (that can only happen when a process suspects q). Thus, in Phase 4, q receives $\lceil\frac{n+1}{2}\rceil$ messages of the type $(-, r, ack)$ (and no messages of the type $(-, r, nack)$), and q R-broadcasts $(q, r, estV_q, -, decide)$ (l.42). By the validity and agreement properties of Reliable Broadcast, eventually all correct processes R-deliver q's message (l.43) and *decide* (l.45)—a contradiction.

So, by Case 2 at least one correct process decides, and by Case 1 all correct processes eventually decide. □

Lemma 13 (Uniform integrity) *Every process decides at most once.*

PROOF. Follows directly from Algorithm 3.2, where no process decides more than once. □

Lemma 14 (Uniform agreement) *No two processes decide differently.*

PROOF. If no process ever decides, the lemma is trivially true. If any process decides, it must have previously R-delivered a message of the type $(-, -, -, -, decide)$ (l.43). By the uniform integrity property of Reliable Broadcast and the algorithm, a coordinator previously R-broadcast this message. This coordinator must have received $\lceil\frac{n+1}{2}\rceil$ messages of the type $(-, -, ack)$ in Phase 4 (l.40). Let r be the smallest round number in which $\lceil\frac{n+1}{2}\rceil$ messages of the type $(-, r, ack)$ are sent to a coordinator in Phase 3 (l.35). Let c denote the coordinator of round r, that is, $c = sysList[((r-1) \bmod n)+1]$. Let $estV_c$ denote c's estimate at the end of Phase 2 of round r. We claim that for all rounds $r' \geq r$, if a coordinator c' sends $estV_{c'}$ in Phase 2 of round r' (l.28), then $estV_{c'} = estV_c$.

The proof is by induction on the round number. The claim trivially holds for $r' = r$. Now assume that the claim holds for all $r', r \leq r' < k$. Let c_k be the coordinator of round k, that is, $c_k = \mathit{sysList}[((k-1) \bmod n) + 1]$. We will show that the claim holds for $r' = k$, that is, if c_k sends estV_{c_k} in Phase 2 of round k (l.28), then $\mathit{estV}_{c_k} = \mathit{estV}_c$.

From Algorithm 3.2 it is clear that if c_k sends estV_{c_k} in Phase 2 of round k (l.28) then it must have received estimates from at least $\lceil \frac{n+1}{2} \rceil$ processes (l.20).[7] Thus, there is some process p such that (1) p sent a (p, r, ack) message to c in Phase 3 of round r (l.35), and (2) $(p, k, \mathit{estV}_p, -, \mathit{ts}_p)$ is in $\mathit{msgs}_{c_k}[k]$ in Phase 2 of round k (l.21). Since p sent (p, r, ack) to c in Phase 3 of round r (l.35), $\mathit{ts}_p = r$ at the end of Phase 3 of round r (l.34). Since ts_p is nondecreasing, $\mathit{ts}_p \geq r$ in Phase 1 of round k. Thus, in Phase 2 of round k, $(p, k, \mathit{estV}_p, -, \mathit{ts}_p)$ is in $\mathit{msgs}_{c_k}[k]$ with $\mathit{ts}_p \geq r$. It is easy to see that there is no message $(q, k, \mathit{estV}_q, -, \mathit{ts}_q)$ in $\mathit{msgs}_{c_k}[k]$ for which $\mathit{ts}_q \geq k$. Let t be the largest ts_q such that $(q, k, \mathit{estV}_q, -, \mathit{ts}_q)$ in $\mathit{msgs}_{c_k}[k]$. Thus, $r \leq t < k$.

In Phase 2 of round k, c_k executes $\mathit{estV}_{c_k} \leftarrow \mathit{estV}_q$ where $(q, k, \mathit{estV}_q, -, t)$ is in $\mathit{msgs}_{c_k}[k]$ (l.26). From Algorithm 3.2, it is clear that q adopted estV_q as its estimate in Phase 3 of round t (l.32). Thus, the coordinator of round t sent estV_q to q in Phase 2 of round t (l.28). Since $r \leq t < k$, by the induction hypothesis, $\mathit{estV}_q = \mathit{estV}_c$. Thus, c_k sets $\mathit{estV}_{c_k} \leftarrow \mathit{estV}_c$ in Phase 2 of round k (l.26). This concludes the proof of the claim.

We now show that, if a process decides a value, then it decides estV_c. Suppose that some process p R-delivers $(q, r_q, \mathit{estV}_q, -, \mathit{decide})$, and thus decides estV_q. By the uniform integrity property of Reliable Broadcast and the algorithm, process q must have R-broadcast $(q, r_q, \mathit{estV}_q, -, \mathit{decide})$ in Phase 4 of round r_q (l.42). From Algorithm 3.2, some process q must have received $\lceil \frac{n+1}{2} \rceil$ messages of the type $(-, r_q, ack)$ in Phase 4 of round r_q (l.41). By the definition of r, $r \leq r_q$. From the above claim, $\mathit{estV}_q = \mathit{estV}_c$. □

Lemma 15 (Uniform validity) *If a process decides v, then v was proposed by some process.*

PROOF. From Algorithm 3.2, it is clear that all *estimates* that a coordinator receives in Phase 2 are proposed values. Therefore, the decision value that a coordinator selects from these *estimates* must be the value proposed by some process. Thus, uniform validity of Lazy Consensus is also satisfied. □

The two properties *proposition integrity* and *weak laziness* are specific to the Lazy Consensus problem. In order to prove them, we first prove some lemmas.

Lemma 16 *Every process that terminates the algorithm considers the same value for the system list sysList after termination.*

PROOF. The proof is a trivial adaptation of Lemma 14 (uniform agreement) to estL. □

[7] Note that $r < k$ hence round k is not the first round.

Lemma 17 *Given a sequence of Lazy Consensus problems, processes begin every instance of the problem with the same system list.*

PROOF. The proof is by induction on the instance number k. If $k = 0$, then all processes initialize the system list using the lexicographical order of processes (l.2). If $k > 0$, then it follows from Lemma 16 that, if the system list is the same for all processes at the beginning of instance $k - 1$, then it is the same for all processes at the beginning of instance k. □

Lemma 18 *For each process p in Π_S, after p changes its estimate $estV_p$ to a value different from $\perp$, then $estV_p \neq \perp$ is always true.*

PROOF. A process p changes the value of its estimate $estV_p$ only at lines 18, 24, 26, and 32. Assuming that $estV_p$ is different from $\perp$, we have to prove that a process p does not set $estV_p$ to $\perp$ if it reaches one of the aforementioned lines.

The result is trivial for lines 18, 24 (by hypothesis the function giv never returns $\perp$) and line 26 (the process selects a value explicitly different from $\perp$).

At line 32, a process sets its estimate to a value received from the coordinator. This value is sent by the coordinator c_p at line 28. Before reaching this line, c_p changed its own estimate $estV_{c_p}$ at one of the following lines: 18, 24, or 26. As shown above, $estV_{c_p}$ is never set to $\perp$ at these lines. □

Lemma 19 *During a round r, a process p proposes a value only if p is coordinator of round r and $estV_p = \perp$.*

PROOF. We say that a process proposes a value when it executes $estV_p \leftarrow$ **eval** giv (line 18 or 24). By line 16, p proposes a value only if p is the coordinator of the round (i.e., $p = c_p$). Let us consider line 18 and line 24 separately.

Line 18: The test at line 17 ensures that line 18 is executed only during the first round. Before executing line 18, $estV_p$ of the coordinator p is trivially equal to $\perp$ (initial value).

Line 24: The result directly follows from the test at line 23. □

Lemma 20 (Proposition integrity) *Every process proposes a value at most once.*

PROOF. We say that a process propose a value when it executes $estV_p \leftarrow$ **eval** giv (lines 18 and 24). We prove the result by contradiction. Assume that some process p proposes a value twice. By definition giv returns a value different from $\perp$. By Lemma 18, once $estV_p \neq \perp$, it remains different from $\perp$ forever. By Lemma 19, p proposes a value only if $estV_p = \perp$. A contradiction with the fact that p proposes a value twice. □

Lemma 21 *If two processes p and q propose a value, then p or q is suspected by a majority of processes in Π_S.*

PROOF. We prove this by contradiction. We assume that neither p nor q are suspected by a majority of processes in Π_S. From Lemma 19 and the rotating coordinator paradigm (there is only one coordinator in each round), p and q do not propose a value in the same round. Let r_p (resp. r_q) be the round in which p (resp. q) proposes a value. Let us assume, without loss of generality, that p proposes before q ($r_p < r_q$).

During round r_p, any process in Π_S either suspects p or adopts p's estimate (lines 30, 31, 32). Since p is not suspected by a majority of processes in Π_S (assumption), a majority of processes adopt p's estimate. By Lemma 18, it follows that (1) a majority of the processes have an estimate different from $\perp$ for any round $r > r_p$.

Consider now round r_q with coordinator q. At line 20, q waits for a majority of estimate messages. From (1), at least one of the estimate messages contains an estimate $estV \neq \perp$. So the test at line 23 returns false, and q does not call giv at line 24. A contradiction with the fact that q proposes a value in round r_q. □

Corollary 22 (Weak laziness) *If two processes p and q propose a value, then p or q is suspected by some processes in Π_S.*

PROOF. Follows directly from Lemma 21. □

Lemma 21 is obviously not necessary to prove the weak laziness property defined in Section 3.3.2. However, as stated in Footnote 5 on page 26, it is interesting to show that our algorithm ensures a property stronger than weak laziness. The property is established by Lemma 21.

Theorem 23 *Algorithm 3.2 solves the weak Lazy Consensus problem using $\Diamond\mathcal{S}$ in asynchronous systems with $f = \lfloor \frac{n}{2} \rfloor$.*

PROOF. The proof of the theorem follows directly from Lemma 12 (termination), Lemma 13 (uniform integrity), Lemma 14 (agreement), Lemma 15 (validity), Lemma 20 (proposition integrity), and Lemma 22 (weak laziness). □

3.4.4 Selected Scenarios for Semi-Passive Replication

Algorithm 3.2 may seem complex, but most of the complexity is due to the explicit handling of failures and suspicions. So, in order to show that the complexity of the algorithm does not make it inefficient, we illustrate typical executions of the semi-passive replication algorithm based on Lazy Consensus using $\Diamond\mathcal{S}$.

We first present the semi-passive replication in the context of a good run (no failure, no suspicion), as this is the most common case. We then show the execution of the algorithm in the context of one process crash.

- *Scenario 1 (good)*
  We call "good run" a run in which no server process crashes and no failure suspicion is generated. The communication pattern of semi-passive replication is illustrated in Figure 3.4. It is noteworthy that the state updates do not appear on the critical path of the client's request (highlighted in gray on the figure).

- *Scenario 2 (crash)*
  We also illustrate the worst case latency for the client in the case of one crash, without incorrect failure suspicions. As illustrated in Figure 3.5 and discussed in Section 3.4.4, the worst case scenario happens when the primary p_1 (i.e., the initial coordinator of the Lazy Consensus algorithm) crashes immediately after processing the client request, but before being able to send the update message *upd* to the backups (compare with Fig. 3.4). In this case, the communication pattern is different from usual algorithm for passive replication in asynchronous systems, as there is no membership change.

Semi-Passive Replication in Good Runs

Let Figure 3.4 represent the execution of Lazy Consensus number k. The server process p_1 is the initial coordinator for consensus k and also the primary. After receiving the request from the

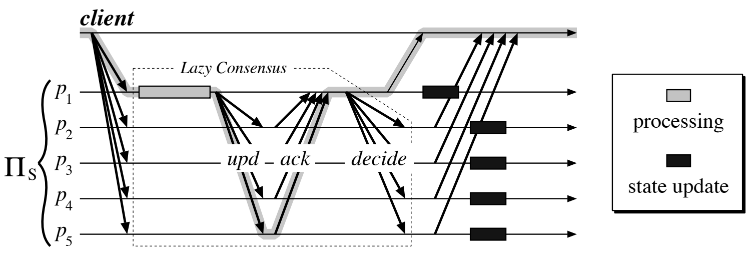


Figure 3.4: Semi-passive replication (good run)

(highlighted in gray: critical path request-response)

client, the primary p_1 handles the request. Once the processing is done, p_1 has the initial value for consensus k. According to the Lazy consensus protocol, p_1 multicasts the update message *upd* to the backups, and waits for *ack* messages. Once *ack* messages have been received (actually from a majority), process p_1 can decide on *upd*, and multicast the *decide* message to the backups. As soon as the *decide* message is received, the servers update their state, and send the reply to the client.

Semi-Passive Replication in the Case of One Crash

In the case of one crash, the execution of the Lazy Consensus algorithm is as follows. If the primary p_1 crashes, then the backups eventually suspect p_1, send a negative acknowledgement message *nack* to p_1 (the message is needed by the consensus algorithm), and start a new round. The server process p_2 becomes the coordinator for the new round, i.e., becomes the new primary,

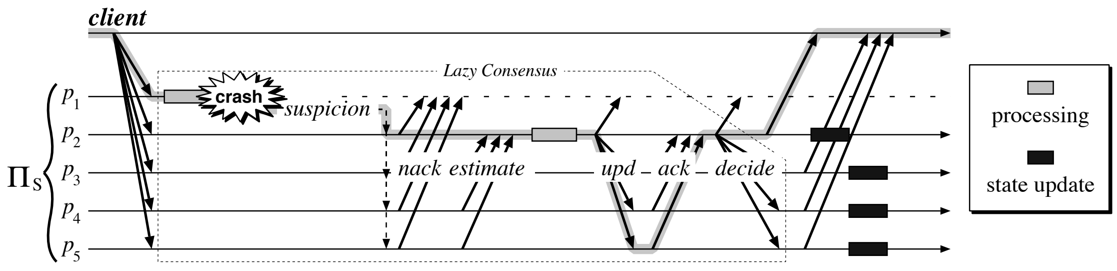


Figure 3.5: Semi-passive replication with one failure (worst case)

(highlighted in gray: critical path request-response)

and waits for $estimate$ messages from a majority of servers: these messages might contain an initial value for the consensus, in which case p_2 does not need to process the client request again. In our worst case scenario, the initial primary p_1 has crashed before being able to multicast the update value upd. So none of the $estimate$ messages received by p_2 contain an initial value. In order to obtain one, the new primary p_2 processes the request received from the client (Fig. 3.5), and from that point on, the scenario is similar to the "good run" case of the previous section (compare with Fig. 3.4).

3.5 Notes on Other Replication Techniques

In this section, we compare semi-passive replication with other replication techniques. More specifically, we discuss the differences and the similarities with four other replication techniques: passive replication, coordinator-cohort, semi-active replication, and active replication.

3.5.1 Passive Replication and Coordinator-Cohort

Passive replication and coordinator-cohort are two replication techniques that are based on the principle that requests should be processed by only one of the replicas. As a result, both techniques rely on the selection of one replica to (1) process requests received from clients, and (2) update the backups after the processing of each request. Aside from a different interaction model between clients and replicas, passive replication and coordinator-cohort have much in common. More specifically, in both techniques the selection of the primary (resp. coordinator) is usually based on a group membership service.

Passive replication Passive replication, also called primary-backup [BMST93], is a well known replication technique. It selects one replica as the *primary*, and the others are *backups*. With this protocol, the client interacts with the primary only. It sends its request to the primary which handles it and sends update messages to the backups. If the primary crashes, a backup becomes the new primary. The client is informed that a new primary has been selected, and must reissue its request.

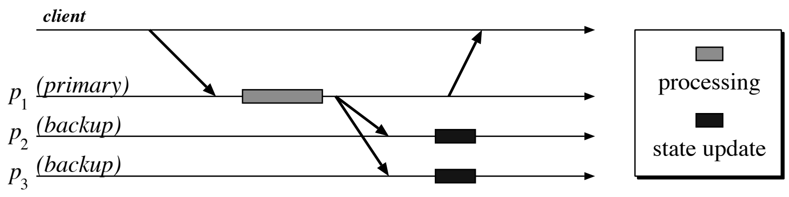


Figure 3.6: Illustration of passive replication.

Coordinator-cohort Coordinator-cohort [BJREA85] is a variant of passive replication implemented in the context of the Isis system [BVR93]. In this replication scheme, one of the replicas is designated as the *coordinator* (primary) and the other replicas as the *cohort* (backups). A client sends its request to the whole group, the coordinator handles the request and sends update messages to the cohort. If the coordinator crashes, a new coordinator is selected by the group and takes over the processing of the request.

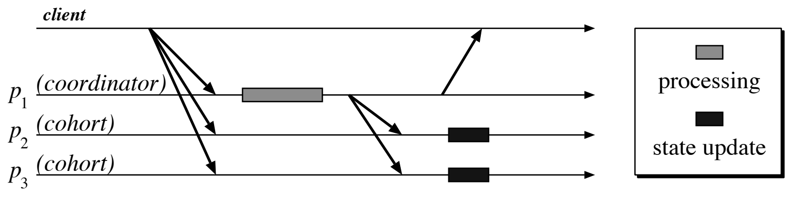


Figure 3.7: Illustration of coordinator-cohort.

3.5.2 Semi-Passive *vs.* Active Replication

In the active replication technique, also called the *state-machine* approach [Sch93], every replica handles the requests received from the client, and sends a reply. In other words, the replicas behave independently and the technique consists in ensuring that all replicas receive the requests in the same order. This technique is appreciated for its low response time, even in the case of a crash. It has however two important drawbacks: (1) the redundancy of processing implies a high resource usage, and more importantly (2) the handling of the requests has to be deterministic.

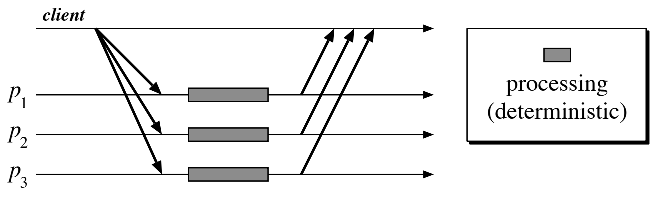


Figure 3.8: Illustration of active replication.

In semi-passive replication, the request is handled by only one replica. As a result, requests do not need to be handled deterministically.

3.5.3 Semi-Passive *vs.* Semi-Active Replication

The semi-active replication technique was implemented in the context of Delta-4 [Pow91], which assumes a synchronous system model. It was developed to circumvent the problem of non-determinism with active replication, in the context of time-critical applications. This technique is based on active replication and extended with the notion of *leader* and *followers*. While the actual processing of a request is performed by all replicas, it is the responsibility of the leader to perform the non-deterministic parts of the processing and inform the followers. This technique is close to active replication, with the difference that non-deterministic processing is made possible.

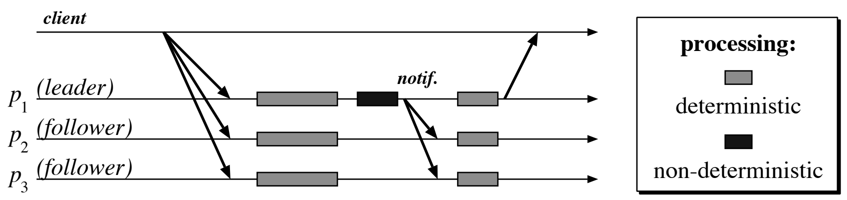


Figure 3.9: Illustration of semi-active replication.

The main difference between semi-passive and semi-active replication is their respective behavior in the ideal case. Indeed, semi-active replication is essentially an active replication scheme because, in the absence of non-deterministic processing, all replicas handle the requests. Conversely, in the absence of crashes and failure suspicions, semi-passive replication ensures that requests are only processed by a single process. This explains why the former replication technique is semi-*active*, whereas the latter is semi-*passive*.

3.6 Group Membership and Semi-Passive Replication

3.6.1 Group Membership Service or not?

The algorithm for semi-passive replication does not rely on a group membership service. Conversely, implementations of passive replication and coordinator-cohort are usually based on a group membership service. What is the significance of this difference? This is discussed now.

Roles of a Membership Service in Replication Techniques

A (primary partition) group membership service serves two purposes in the context of passive replication and coordinator-cohort: *management of the composition of the group* and *selection of the primary*.

Management of the composition of the group An obvious role of a group membership service is the management of the composition of group. The membership service maintains at any time a list of members of the group (called the current *view* of the group). Processes can explicitly join or leave the group, and processes that are suspected to have crashed are automatically removed from the group.

Managing the composition of groups is indeed an important role of a group membership service. For example, from a practical point of view, it is necessary to allow processes to join the group during the lifetime of the system, e.g., to replace processes that have crashed.

Selection of the primary Passive replication and semi-passive replication are both based on the existence of a primary process. One of the major difficulties of these replication techniques is the selection of the primary.

In many systems, the selection of the primary is based on the current view of the group of replicas (e.g., Isis [BVR93], Horus [VRBG$^+$95], Totem [MMSA$^+$95], or Phoenix [Mal96]): the primary is obtained by applying some deterministic function to the current view. For instance, after a view change, the first process in the lexicographical order becomes the primary.

Performance Tradeoff

Using a membership service for both the composition of the group and the selection of the primary poses some problems. Despite the architectural simplicity of this approach, the cost of incorrect suspicions introduces a difficult performance tradeoff between two conflicting goals: rare occurrences of incorrect suspicions, and a fast reaction to actual failures.

The tradeoff is related to the difficulty, in asynchronous systems, to distinguish a crashed process from a very slow one. Because of this, a failure detection mechanism can follow one of two opposing policies: *aggressive* or *conservative*. When a process does not answer, a *conservative policy* assumes that the process is just slow (i.e., conservative timeout). Conversely, an *aggressive policy* quickly assumes that the process has crashed (i.e., aggressive timeout). A conservative policy generates fewer incorrect suspicions, but an aggressive policy leads to a quicker reaction to failures.

Unfortunately, an aggressive policy leads to frequent incorrect failure suspicions. This has a high price as it results in the removal of the suspected process from the group. In addition to launching the membership algorithm uselessly, the removed process will ask to rejoin the group shortly after. As a result, the membership algorithm is executed a second time, together with a state transfer. Running the membership algorithm is a costly operation, but the greatest cost is largely due to the state transfer. Indeed, from our experience, a state transfer is a delicate operation that can prove extremely costly in a real-world environment [DMS98].

Instability of Group Membership

The automatic exclusion of processes poses some more problems. In addition to the problem mentioned above, this approach leads to a practical problem: cascading. In most systems, the failure detection mechanism relies on communication and timeouts, and hence is normally quite sensitive to an increased load on the communication medium. Also, each incorrect suspicion may lead to two executions of the view change protocol, plus a state transfer. This introduces the following feedback loop: incorrect suspicions increase the communication load, which potentially increases the number of incorrect suspicions. Without being an expert in control theory, it is easy to see that introducing such a loop makes the whole system potentially unstable. This may compel the system to exclude all replicas, thus leading to an obvious liveness problem.

There are different solutions to either solve, or at least reduce the potential instability of the system:

1. Decoupling the failure detection from other traffic on the communication medium provides a way to break the feedback loop. This can be done either (a) by relying on an independent communication medium for failure detection messages, or (b) by using traffic reservation techniques and high priority messages for failure detection. The drawback is that this method requires a specific environment (hence lacks generality), and this does not fully solve the problem as interdependencies remain at the scheduling level. For instance, multiple threads may unexpectedly delay failure detection messages.

2. Using a conservative failure detection mechanism reduces the sensitivity to the load. The probability of incorrect suspicions is thus kept near zero, and there are little risks that the system gets engaged into the loop. No matter how unlikely, the risk nevertheless remains.

3. Reducing the additional communication induced by each incorrect suspicion decreases the effect of those suspicions on the load of the system. Similar to the previous point, this does not quite solve the problem, but only reduces the chance that anything bad will occur.

Eventual Exclusion of Crashed Members

The assumption that the group of processes is static and that only a limited number of them eventually crash is not always acceptable in practice. From a practical standpoint, this would mean that the reliability of the system decreases until the system eventually ceases to exist.

There are two possible approaches to avoid the fateful degradation of the system. The first approach considers algorithms in a model where processes can crash and recover (e.g., [OGS97, ACT98, HMR98]). The second approach relies on the principle that every crashed process is eventually replaced by a new live processes (e.g., [BCG91]).

A group membership service can easily provide the mechanisms necessary for the replacement of crashed processes. Indeed, when a process is crashed, it is excluded from its process group. Then, the application only has to provide a new live process to replace the crashed one, thus reverting the group to its initial level of reliability.

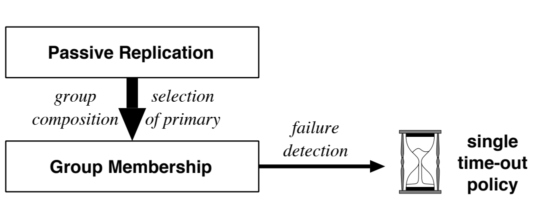
This of course requires that crashed processes are detected and eventually excluded from the group. In this case, there is however no real motivation for a prompt detection of a crashed process, since it is very unlikely that a second failure occurs during that period.[8]

Separation of Concern

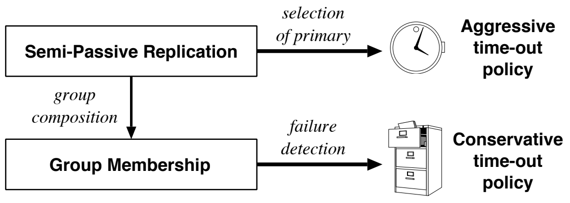
There is a clear contradiction between the need for a conservative failure detection mechanism to ensure stability, and an aggressive one to ensure a good responsiveness in the case of failure.

[8]This assumes that genuine failures occur independently, and on extremely rare occasions (Assumption 2, page 5).

In a system where the selection of the primary and the composition of the group are combined there is room for only one failure detection mechanism (see Fig.3.10(a)). This allows for only one policy, and leads to the following tradeoff. On the one hand, the selection of a primary may require a fast reaction to failures, thus taking advantage of an aggressive policy. On the other hand, tracking the composition of the group does not usually require a fast reaction to failures,[9] but benefits from the extreme stability (very few incorrect suspicions) that a conservative policy offers.



(a) Passive replication (same timeout policy)



(b) Semi-passive replication (different timeout policies)

Figure 3.10: Group composition *vs.* selection of the primary

With semi-passive replication, the group composition and the selection of the primary are managed separately (see Fig.3.10(b)). The algorithm presented in this chapter does not rely on a membership service, and is based on a lightweight mechanism (i.e., the rotating coordinator paradigm). In fact, such a service (based on extremely conservative timeouts) could easily be added to semi-passive replication to manage the composition of the group in order, for instance, to replace crashed processes.

The solution that we advocate consists in separating the selection of the primary from the composition of the group. As a direct consequence, incorrectly suspecting the primary does not trigger its ejection from the group, thus significantly reducing the price of such a suspicion. This makes it possible to design a system in which the group composition uses a conservative failure detection policy, while the selection of the primary relies on an aggressive one.

3.6.2 Undoable Operations at the Primary

We can consider the following scenario. The primary/coordinator p_1 handles a request and modifies its state accordingly, but is (erroneously) suspected before it sends its update message.

Passive replication or coordinator-cohort With passive replication or coordinator-cohort, a view change is launched and p_1 is removed from the new view. A new primary p_2 takes over, handles the request (differently), and sends its own update message to the other processes. Shortly after, p_1 joins back the group and performs a *state transfer*. In other words, the processing of p_1 is undone by the state transfer.

[9]The replacement of the crashed process by a new process rarely requires to be done within a very short time-frame

Semi-passive replication In semi-passive replication, a second coordinator p_2 handles the request (differently) and sends its own update message to the other processes. Since p_1 remains in the group, it receives the update message from p_2 and must undo the modifications to its state in order to be able to apply the new update message. In other words, changes made by the coordinator when handling the request are tentative and must be undoable.[10] The changes only become permanent when the replicas (including the coordinator) execute the update message. This can be implemented using techniques, such as write-ahead logs [Gra79] or shadow pages [LS76].

3.7 Summary

Contribution Semi-passive replication is a replication technique which neither relies on a group membership for the selection of the primary, nor on process controlled crash. While retaining the essential characteristics of passive replication (i.e., non-deterministic processing and restrained use of processing resources), semi-passive replication can be solved in an asynchronous system using a $\Diamond\mathcal{S}$ failure detector.

The semi-passive replication algorithm proposed in this dissertation is based on solving the problem of Lazy Consensus defined in this chapter. Lazy Consensus extends the usual definition of the Consensus problem with a property of Laziness. This additional property is the key to the restrained use of resources in semi-passive replication. The semi-passive replication algorithm only relies on the usual properties of Consensus for its correctness and to handle non-deterministic processing.

Semi-passive replication is a passive replication scheme based on a variant of Consensus (Lazy Consensus). One of the main goals of this chapter is to clarify the relation between passive replication and the Consensus problem in asynchronous distributed systems. It should also help to understand the tradeoffs involved in this class of systems.

Advantages of semi-passive replication One of the main advantages of semi-passive replication is that it avoids unnecessary state transfer operations. This is significant for some real-world applications as the size of a state can be pretty big and thus a state transfer can bear a large impact on the normal execution of the application. An evaluation done for the Swiss stock exchange gives a good illustration of this issue [DMS98].

Unlike passive replication and coordinator-cohort, semi-passive replication allows for an aggressive failure detection policy. In fact, Sergent [Ser98] has shown, using simulation, that the ideal timeout value for a Consensus algorithm is in the order of the average time it takes to execute this algorithm in a failure-free case (i.e., a few milliseconds in a LAN). The performance measurements done to illustrate the FLP impossibility result [UDS00b] confirm Sergent's observation, but also show the difficulty to find an optimal value for such a timeout, even in a LAN.

From the standpoint of a client, the interaction with the replicated server is exactly the same whether the server is replicated using active or semi-passive replication. Furthermore, both tech-

[10]Note that a support for undoable operations is also needed with passive replication if the processing of a request may have a side-effect, such as displaying some information on a screen.

niques can be built on top of a Consensus algorithm (in fact, Lazy Consensus). This is an interesting aspect allowing both replication techniques to coexist in the same system [FDES99].

Limitations Semi-passive replication can reduce the response time of a replicated server in the case of failure, but this response time can still be quite longer than in a failure-free case. Indeed, semi-passive replication makes it possible to reduce the *blackout period* that is associated with the detection of a crash. However, if the primary crashes shortly after processing a request, that request may have to be processed again by a new primary thus at least doubling the total response time. Unlike the detection of failures, this behavior is inherent to the restrained use of resource, and hence cannot be avoided. Nevertheless, the advantage of semi-passive replication over passive replication is still particularly blatant for requests that require a short to moderately short processing time (e.g., up to about a minute), where a long timeout accounts for a large part of the overall response time.

Decoupling the replication algorithm from the group membership service reduces the risk of instabilities in the system. This risk still exists in semi-passive replication, although it is considerably smaller. In fact, breaking the tradeoff allows for choosing a much smaller timeout, as close as possible to the minimum value. This leads to the problem of tuning this timeout in such a way that the aggressive failure detector generates an “acceptable” rate of incorrect suspicions. In practice, this means that timeout values must be tuned when a system is deployed if it can be assumed that it is stable. If the characteristics of the system may vary over time, then aggressive failure detection should rely on adaptive failure detectors (e.g., [CTA00, Che00]).

Extensions The Lazy Consensus algorithm presented in this chapter is adapted from Chandra and Toueg’s Consensus algorithm using $\Diamond\mathcal{S}$ [CT96]. Many other Consensus algorithms could also be adapted to solve Lazy Consensus (e.g., [Sch97, HMRT99, YT95, YST94]). Semi-passive replication is intrinsically asymmetrical—it relies on a primary. Algorithm 3.1 appears to be symmetrical (i.e., no process plays a particular role) only because it relies on the asymmetry of the Lazy Consensus. As a result, Consensus algorithms that are symmetrical (e.g., [ADLS94]) can probably not be adapted to solve Lazy Consensus.

Chapter 4

Classification of Totally Ordered Broadcast and Multicast Algorithms

The men of experiment are like the ant, they only collect and use;
the reasoners resemble spiders, who make cobwebs out of their own substance.
But the bee takes the middle course:
it gathers its material from the flowers of the garden and field,
but transforms and digests it by a power of its own.

— **Francis Bacon** (1561–1626)

There exists a considerable amount of literature on total order broadcast and multicast, and many algorithms have been proposed (more than fifty!), following various approaches. It is however difficult to compare them as they often differ with respect to their actual properties, assumptions, objectives, or other important aspects. It is hence difficult to know which solution is best suited given a certain application context. When confronted to new requirements, the absence of a road map to the problem of total order multicast has often led engineers and researchers to either develop a new algorithm rather than adapt an existing solution (thus reinventing the wheel), or use a solution poorly suited to the application needs. An important step to improve the present situation is to provide a classification of existing algorithms.

Related work Previous attempts have been made at classifying and comparing total order multicast algorithms [Anc93b, AM92, CdBM94, FVR97, May92]. However, none is based on a comprehensive survey of existing algorithms, and hence they all lack generality.

The most complete comparison so far is due to Anceaume and Minet [Anc93b, AM92], who take an interesting approach based on the *properties* of the algorithms. The paper raises some fundamental questions upon which our work draws some of its inspiration. It is however a little outdated now, and slightly lacks generality. Indeed the authors only study seven different algorithms among which, for instance, none is based on a communication history approach (see Sect. 4.3.1).

Cristian, de Beijer, and Mishra [CdBM94] take a different approach focusing on the implementation of those algorithms rather than their properties. They study four different algorithms, and compare them using discrete event simulation. They find interesting results regarding the respective performance of different implementation strategies. Nevertheless, they fail to discuss the respective properties of the different algorithms. Besides, as they compare only four algorithms, this work is less general than Anceaume's.

Friedman and Van Renesse [FVR97] study the impact that packing messages has on the performance of algorithms. To this purpose, they study six algorithms among which those studied by Cristian et al. [CdBM94]. They measure the actual performance of those algorithms and confirm the observations made by Cristian et al. [CdBM94]. They show that packing message indeed provides an effective way to increase the performance of algorithms. The comparison also lacks generality, but this is quite understandable as this is not the main concern of that paper.

Mayer [May92] defines a framework in which total order broadcast algorithms can be compared from a performance point of view. The definition of such a framework is an important step towards an extensive and meaningful comparison of algorithms. However, the paper does not go so far as to actually compare the numerous existing algorithms.

Classification A classification system is based on similarities. Entities are grouped into classes which are defined according to some criteria. Ideally, a perfect classification system would define its classes so that (1) entities with similar characteristics are grouped together, (2) every entity belongs to a class, (3) no entity belongs to two different classes unless one is a subclass of the other, and (4) entities are evenly distributed among classes. But, most importantly, the purpose of any classification is to bring a better understanding on how the various entities relate to each other. This is achieved by carefully choosing the criteria on which the classes are based. Some choices must sometimes be arbitrary, and hence rely on a subjective perception of the field. Nevertheless, any classification system is adequate and serves its purpose if it can bring this understanding.

Contribution In this chapter, we propose a classification system based on the mechanisms of the algorithms, as well as a set of characteristics and assumptions. Based on this classification, we present a vast survey of published algorithms that solve the problem of total order multicast. We separate the presentation between algorithms which order messages using physical time and those which do not use physical time. We however restrict our study to algorithms that neither require specific hardware (unlike, e.g., [CFM87, Jal98, MA91a]) nor a specific network infrastructure or topology (unlike, e.g., [CL96, CMMS96, FM90, Tse89]).

4.1 Specification of Total Order Multicast

When talking about a problem, the first thing to discuss is its specification. Indeed, the specification of a problem *defines* that problem and hence must be considered before anything else. More specifically, the specification of a problem should be considered before considering any algorithm that solves that problem. This is particularly important given that two algorithms that meet even slightly different specifications actually solve different problems.

According to this policy, we first present the specification of the problem of total order broadcast. We then situate that problem with respect to the hierarchy of broadcast problems presented by Hadzilacos and Toueg [HT93].

4.1.1 Total Order Multicast

The problem of Total Order Multicast is defined by four properties: Validity, Agreement, Integrity, and Total Order.

(VALIDITY)
If a correct process broadcasts a message m, then some correct process in $Dest(m)$ eventually delivers m.

(UNIFORM AGREEMENT)
If a process delivers a message m, then all correct processes in $Dest(m)$ eventually deliver m.

(UNIFORM INTEGRITY)
For any message m, every process p delivers m at most once, and only if (1) m was previously broadcast by $sender(m)$, and (2) p is a process in $Dest(m)$.

(UNIFORM TOTAL ORDER)
If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

The well-known problem of Atomic Broadcast[1] (e.g., [HT93, HT94]) appears as a special case of Total Order Multicast, where $Dest(m)$ is equal to Π for all messages. The distinction between broadcast and multicast is further discussed in Section 4.2.3.

4.1.2 Hierarchy of Non-Uniform Problems

The properties of agreement, integrity, and total order that are given above are uniform. This means that these properties not only apply to correct processes but also to faulty processes. Uniform properties are required by some classes of application such as atomic commitment. However, for some other applications uniformity is not necessary. Since enforcing uniformity in an algorithm often has a cost in terms of performance, it is also important to consider weaker problems specified using the following three non-uniform counterparts of the properties mentioned above.

(AGREEMENT)
If a ***correct*** process delivers a message m, then all correct processes in $Dest(m)$ eventually deliver m.

(INTEGRITY)
For any message m, every ***correct*** process p delivers m at most once, and only if (1) m was previously broadcast by $sender(m)$, and (2) p is a process in $Dest(m)$.

[1]Note that the term *Atomic Broadcast* has sometimes been used abusively to designate a broadcast primitive without ordering property [End99, JEG99, GHP83]

(TOTAL ORDER)

If ***correct*** processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

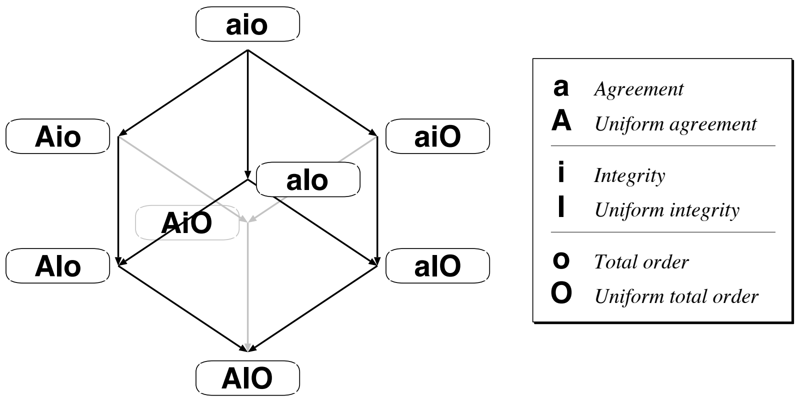


Figure 4.1: Hierarchy of fault-tolerant specifications for total order multicast.

The combinations of uniform and non-uniform properties define eight different specifications to the problem of fault-tolerant total order multicast. Those definitions form a hierarchy of problems. Wilhelm and Schiper [WS95] propose a hierarchy based on two properties (total order and agreement) thus resulting in four different problems. In this chapter, we complement this hierarchy by considering the property of uniform versus non-uniform integrity. We illustrate in Figure 4.1 the hierarchy given by the eight combinations of agreement, integrity, and total order.

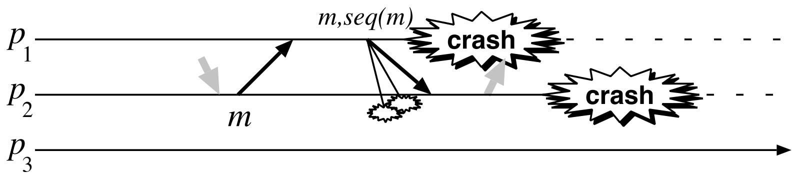


Figure 4.2: Violation of Uniform Agreement (example)

Figure 4.2 illustrates a violation of the Uniform Agreement with a simple example. In this example, process p_2 sends a message m to a sequencer p_1, which gives a sequence number to m and then relays m to all processes. However, p_1 crashes while it is relaying m, in such a way that only p_2 receives the message. Upon reception of m and the sequence number, p_2 delivers m and then crashes. As a result, the system is in a situation where the message m has been delivered by some process (p_2), but no correct process (e.g., p_3) will ever be able to deliver it. Indeed, m is lost to the system because every process that has a knowledge of m (p_1 and p_2) has already crashed.

Uniform Agreement and message stabilization In algorithms that ensure Uniform Agreement, a process p is not allowed to deliver a message m before that message is stable. A message m is *stable* once it is certain that m will eventually be delivered by all correct processes.

Uniform properties and Byzantine failures It is important to note that algorithms tolerant to Byzantine failures can guarantee none of the uniform properties. This is understandable as no behavior can be enforced on Byzantine processes. In other words, nothing can prevent a Byzantine process from (1) delivering a message more than once (violates integrity), (2) delivering a message that is not delivered by other processes (violates agreement), or (3) delivering two messages out of order (violates total order).

A note on Integrity Except in a system with Byzantine processes, Uniform Integrity is easily ensured.[2] For this reason, almost every algorithm found in the literature satisfies Uniform Integrity rather than its weaker counterpart. Hence, in the sequel, we hardly discuss Integrity.

4.1.3 Problem of Contamination

The specification of total order broadcast can be either uniform or not, depending whether it specifies the behavior of faulty processes or not. However, even with the strongest specification (Uniform Agreement, Uniform Integrity, and Uniform Total Order) a faulty process is not prevented from getting an inconsistent state. This becomes a serious problem when one considers that this process can multicast a message based on this inconsistent state, and hence contaminate correct processes [GT91, AM92, Anc93b, HT94].

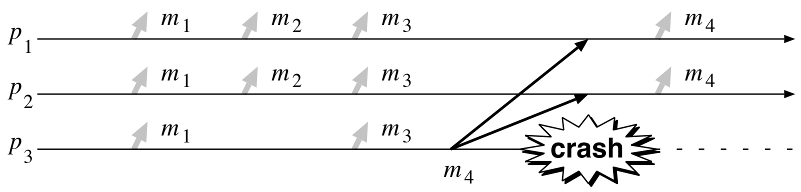


Figure 4.3: Contamination of correct processes (p_1, p_2) by a message (m_4) based on an inconsistent state (p_3 delivered m_3 but not m_2).

Illustration Figure 4.3 illustrates an example [CBPD99, HT94, HT93] where an incorrect process contaminates the correct processes by sending a message based on an inconsistent state. Process p_3 delivers messages m_1 and m_3, but not m_2. It then has an inconsistent state when it broadcasts m_4 to the other processes and then crashes. The correct processes p_1 and p_2 deliver m_4, thus getting contaminated by the inconsistent state p_3 had before it crashed (p_3 delivered m_3 but not m_2). It is important to stress again that the situation depicted in Figure 4.3 satisfies the strongest specification.

Specification The problem of contamination is due to a gap in the usual specification. It is then necessary to extend the specification of total order multicast in order to prevent this situation from happening. The specification can prevent contamination in two ways. One way is to forbid

[2]Ensuring Uniform Integrity requires (1) to uniquely identify messages, and (2) to keep track of delivered messages.

faulty processes from sending messages if their state is inconsistent. This is however difficult to formalize as a property. Hence the second (and stronger) solution is usually preferred, which consists in preventing any process from delivering a message that may lead to an inconsistent state [AM92, KD00, CdBM94].

(PREFIX ORDER)
For any two processes p and q, either $hist(p)$ is a prefix of $hist(q)$ or $hist(q)$ is a prefix of $hist(p)$, where $hist(p)$ and $hist(q)$ are the histories of messages delivered by p and q respectively.

Algorithms Among the numerous algorithms studied in the chapter, a large majority ignores the problem of contamination in their specification. In spite of this, some of these algorithms are implemented in such a way that contamination can never occur. The mechanism of these algorithms either (1) prevents all processes from reaching an inconsistent state, or (2) prevents processes with an inconsistent state from sending messages to other processes.

Contamination and Byzantine failures Contamination is impossible to avoid in the context of arbitrary failures, because a faulty process may be inconsistent even if it delivers all messages correctly. It may then contaminate the other processes by multicasting a bogus message that seems correct to every other process [HT94].

4.2 Architectural Properties

When a total order multicast algorithm is used in some application context, the architectural properties of this algorithm are very important. For instance, there is a general distinction between *broadcast* algorithms (messages are received and delivered by all processes in the system) and *multicast* algorithms (messages are delivered only by a subset of the processes in the system). In this section, we define the following architectural properties of algorithms: single *vs.* multiple destination groups, and closed *vs.* open groups.

4.2.1 Single *vs.* Multiple Destination Groups

Most algorithms present in the literature assume that there is only one destination group in the system. Since the ability to multicast messages to multiple (overlapping) groups is required by some applications, we consider it as an important architectural characteristic of an algorithm.

Single group ordering Messages are sent to only one group of destination processes and all messages that are sent to this group are delivered in the same order. This class groups a vast majority of the algorithms that are studied in this chapter. Single group ordering can be defined by the following property:

$$\forall m \in \mathcal{M} : Dest(m) = \Pi_{dest}$$

Even though these algorithms can be adapted fairly easily to systems with multiple *non-overlapping* groups, they often cannot be made to cope decently with groups that overlap.

Multiple groups ordering Multiple groups ordering occurs when multiple destination groups exist in the system, that may overlap. An algorithm designed for such a system must guarantee that messages are delivered in a total order by all destination processes, including those that are at the intersection of two groups. For instance, if two messages m_A and m_B are sent to two overlapping groups A and B respectively, then m_A and m_B must be delivered in the same relative order by all processes that belong to the intersection $A \cap B$.

Trivially, any algorithm that solves the problem of total order broadcast in the context of a single group can solve it for multiple groups. Indeed, one can form a super-group with the union of all groups in the system. Whenever a message is multicast to a group, it is instead broadcast to the super-group, and processes for which this message is not destined simply discard it. To avoid such a trivial solution, we require multiple groups ordering algorithms to satisfy the following minimality property.

(WEAK MINIMALITY)
The execution of the algorithm implementing the total order multicast of a message m to a destination set $\mathit{Dest}(m)$ involves only $\mathit{sender}(m)$, the processes in $\mathit{Dest}(m)$, and possibly an external process p_s (where $p_s \neq \mathit{sender}(m)$ and $p_s \notin \mathit{Dest}(m)$).

Genuine multicast A genuine multicast is defined by Guerraoui and Schiper [GS97c] as a multiple groups ordering algorithm which satisfies the following minimality property.

(STRONG MINIMALITY)
The execution of the algorithm implementing the total order multicast of a message m to a destination set $\mathit{Dest}(m)$ involves only $\mathit{sender}(m)$, and the processes in $\mathit{Dest}(m)$.

4.2.2 Closed *vs.* Open Groups

In the literature, many total order multicast algorithms are designed with the implicit assumption that messages are sent *within* a group of processes. This originally comes from the fact that early work on this topic was done in the context of highly available storage systems [CASD84]. However, a large part of distributed applications are now developed by considering more open interaction models, such as the client-server model, N-tier architectures, or publish/subscribe. For this reason, we consider that it is important for a process to be able to multicast messages to a group it does not belong to. Consequently, it is an important characteristic of algorithms to be easily adaptable to accommodate open interaction models.

Closed group Closed group algorithms require sending processes to be part of the destination processes.

$$\forall m \in \mathcal{M}\ (\mathit{sender}(m) \in \mathit{Dest}(m))$$

As a result, these algorithms do not allow external processes (processes that are not member of the group) to multicast messages to the destination group. This limitation forbids the use of such algorithms in the context of large-scale systems in which client processes are short-lived and their number is unbounded (e.g., on-line reservation systems, Internet banking).

Open group Conversely, open group algorithms allow any arbitrary process in the system to broadcast messages to a group, whether that process belongs to the destination group or not. Open group algorithms are more general than their closed group counterparts because the former can be used in a closed group architecture while the opposite is not true.

Remarks Most algorithms considered in this chapter are designed for closed groups. However, in the sequel, we classify into the open group category all algorithms that can *trivially* be adapted to allow external processes to multicast to the group. By trivially adapted, we mean that such an adaptation requires (1) no additional communication step, (2) no additional information to manage (e.g., longer vector clocks), and (3) no additional computation in the algorithm.

For instance, condition (1) rules out the following transformation from a closed group to an open group algorithm. An external sending process sends its message m to a member p of the group, which then broadcasts m within the group. This solution has the drawback that the client must monitor p, and resend m if p has crashed. This also means that m can be received multiple times by the members of the group, and thus duplicates must be detected and discarded.

Another possibility would be to include the sending process in the group and require it to discard the messages that it receives. This approach would be against condition (3), and probably also condition (2).

4.2.3 Other Architectural Properties

Broadcast *vs.* Multicast

A *broadcast* primitive is defined as one that sends messages to all processes in a system (i.e., single closed group).

$$\text{Broadcast: } \forall m \in \mathcal{M}\ (Dest(m) = \Pi)$$

In contrast, a *multicast* primitive sends messages to any subset of the processes in the system (i.e., open multiple groups). However, the distinction between broadcast and multicast is not precise enough, because it mixes two orthogonal aspects; single *vs.* multiple destination groups, and closed *vs.* open group.

Source Ordering

Some papers (e.g., [GMS91, Jia95]) make a distinction between single source and multiple sources ordering. These papers define single source ordering algorithms as algorithms that ensure total order only if a *single* process broadcasts every messages. This problem is actually a simplification of FIFO broadcast and is easily solved using sequence numbers. In this chapter, we are interested in algorithms that ensure total order and not only FIFO order. Hence we do not discuss this issue further, and all algorithms presented here provide multiple sources ordering:

$$|\Pi_{sender}| \geq 1$$

4.3 Ordering Mechanisms

In this section, we propose a classification of total order multicast algorithms in the absence of failures. The first question that we ask is: “who builds the order?”. More specifically, we are interested in the entity which generates the information that is necessary for defining the order of messages (e.g., timestamp or sequence number).

We classify the processes in three distinct categories according to the role they take in the algorithm: sender process, destination process, or external process. A *sender process*, or *sender*, is a process p from which a message originates ($p \in \Pi_{sender}$). A *destination process*, or *destination*, is a process p to which a message is destined ($p \in \Pi_{dest}$). Finally, an *external process* is some process p that is not necessarily a sender nor a destination ($p \in \Pi$). When an external process creates the order, it is called a *sequencer*. A single process may combine these different roles (e.g., sender *and* destination). However, we represent these roles separately as they are distinct from a *conceptual* viewpoint.

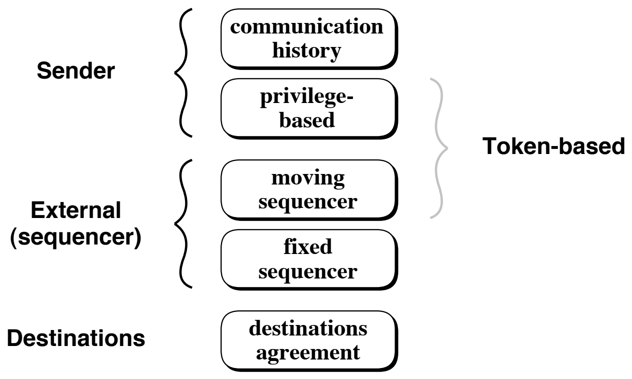


Figure 4.4: Classes of total order multicast algorithms.

Following from the three different roles that processes can take, total order multicast algorithms can fall into three basic classes, depending whether the order is built by sender, external, or destination processes respectively. Among those three basic classes, there exist differences in the algorithms. This allows us to define subclasses as illustrated in Figure 4.4. This results in the five classes illustrated on the figure: *communication history*, *privilege-based*, *moving sequencer*, *fixed sequencer*, and *destinations agreement*. Algorithms that are either privilege-based or based on a moving sequencer are commonly referred to as token-based algorithms.

The terminology used by this classification is partly borrowed from other authors. For instance, “communication history” and “fixed sequencer” were proposed by Cristian and Mishra [CM95].

The term “privilege-based” was suggested by Malkhi.[3] Finally, Le Lann and Bres [LLB91] group algorithms into three classes based on where the order is built. However, the definition of the classes is specific to a client-server architecture.

In this section, we present the five classes of algorithms. We illustrate each class with a simple algorithm for which we give the pseudo-code. These algorithms are presented for the sole purpose of illustrating the corresponding category. Although they are inspired from existing algorithms, they are simplified. Also, as mentioned earlier, the problem of failures is ignored in this section, thus none of these algorithms tolerate failures.

4.3.1 Communication History

Communication history algorithms are based on the following principle. A partial order “$<$” is defined on messages, based on the history of the communication. Communication history algorithms deliver messages in a total order that is compatible with the partial order “$<$”. This class of algorithms takes its name from the fact that the order is induced by the communication. The partial order ”$<$” is defined as follows:

$$\mathit{send}(m) \longrightarrow \mathit{send}(m') \Rightarrow m < m'$$

Where “$\longrightarrow$” denotes Lamport’s relation “happened before” [Lam78] (see Sect. 2.3.1).

Algorithm 4.1 is typical of this class. It is inspired by Lamport [Lam78] and based on his logical clocks.[4] In short, Lamport’s logical clocks are defined according to the relation “$\longrightarrow$” (happened before), in such a way that the following condition holds: Let $ts(m)$ be the logical timestamp of message m. Lamport’s clocks ensure that

$$\mathit{send}(m) \longrightarrow \mathit{send}(m') \Rightarrow ts(m) < ts(m')$$

So, in order to deliver the messages in a total order compatible with “$<$”, it is sufficient to deliver the messages in a total order compatible with $ts(m)$ as follows: if $ts(m) < ts(m')$ then message m has to be delivered before message m'. Messages with the same logical timestamp are delivered according to an arbitrary order based on their sender, and denoted by $\prec$. The total order relation on events “$\Longrightarrow$” is then defined as follows: if m and m' are two messages, then $m \Longrightarrow m'$ if and only if either (1) $ts(m) < ts(m')$ or (2) $ts(m) = ts(m')$ and $\mathit{sender}(m) \prec \mathit{sender}(m')$.

Consider Algorithm 4.1. When a process wants to multicast a message m, it sends m to all processes with a Lamport timestamp. Upon reception of such a timestamped message m, processes store m as a received yet undelivered message. A process can deliver message m only after it knows that no other process can multicast a new message with a timestamp lower or equal to $ts(m)$. Messages are then delivered according to the total order relation “$\Longrightarrow$”.

[3] *Private communications:* Dahlia Malkhi pointed out that the previously used term “permission-based algorithms” entered in conflict with the taxonomy used for mutual exclusion algorithms [Ray91]. As permission-based mutual exclusion algorithms are based on a different concept (e.g., [RA81]), using the same terminology here would then lead to confusion.

[4] Algorithm 4.1 can also be seen as a simplification of the Newtop algorithm [EMS95].

Observation 2
Algorithm 4.1 is not live (some messages may never get delivered). To overcome this problem, communication history algorithms proposed in the literature usually force processes to send messages.

Observation 3
In synchronous systems, communication history algorithms use physical timestamps and rely on synchronized clocks. The nature of such systems makes it unnecessary to force processes to send messages in order to guarantee the liveness of the algorithm. This can be seen as an example of the use of time to communicate [Lam84].

4.3.2 Privilege-Based

Privilege-based algorithms rely on the idea that senders can multicast messages only when they are granted the privilege to do so. The arbitration between senders makes it possible to totally order the messages as they are sent. Building the total order requires to solve the problem of FIFO broadcast (easily solved with sequence numbers at the sender), and to ensure that passing the privilege to the next sender does not violate this order.

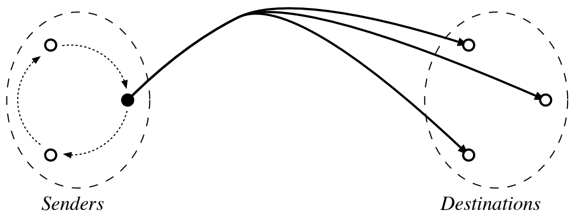


Figure 4.5: privilege-based algorithms.

Figure 4.5 illustrates this class of algorithms. The order is defined by the senders when they multicast their messages. The privilege to multicast (and order) messages is granted to only one process at a time, but this privilege circulates from process to process among the senders.

Algorithm 4.2 illustrates the principle of privilege-based algorithms. Messages receive a sequence number when they are multicast, and the privilege to multicast is granted by circulating a token message among the senders.

Senders circulate a token message that carries a sequence number for the next message to multicast. When a process wants to multicast a message m, it must first wait until it receives the token message. Then, it assigns a sequence number to each of its messages and sends them to all destinations. The sender then updates the token and sends it to the next sender. Destination processes deliver messages in increasing sequence numbers.

Observation 4
In privilege-based algorithms, senders usually need to know each other in order to circulate the privilege. This constraint makes privilege-based algorithms poorly suited to open groups, unless there is a fixed and previously known set of senders.

Algorithm 4.1 Simple communication history algorithm.

```
Senders and destinations (code of process $p$):
  Initialisation:
    $received_p \leftarrow \emptyset$                                        {Messages received by process $p$}
    $delivered_p \leftarrow \emptyset$                                       {Messages delivered by process $p$}
    $LC_p[p_1 \dots p_n] \leftarrow \{0, \dots, 0\}$          { $LC_p[q]$: logical clock of process $q$ as seen by process $p$}
  procedure $TO\text{-}multicast(m)$                                   {To TO-multicast a message $m$}
    $LC_p[p] \leftarrow LC_p[p] + 1$
    $ts(m) \leftarrow LC_p[p]$
    send $(m, ts(m))$ to all
  when receive $(m, ts(m))$
    $LC_p[p] \leftarrow \max(ts(m), LC_p[p]) + 1$
    $LC_p[sender(m)] \leftarrow ts(m)$
    $received_p \leftarrow received_p \cup \{m\}$
    $deliverable \leftarrow \emptyset$
    for each message $m'$ in $received_p \setminus delivered_p$ do
      if $ts(m') < \min_{q \in \Pi} LC_p[q]$ then
        $deliverable \leftarrow deliverable \cup \{m'\}$
    deliver all messages in $deliverable$, according to the total order $\Longrightarrow$ (see Sect. 4.3.1)
    $delivered_p \leftarrow delivered_p \cup deliverable$
```

Algorithm 4.2 Simple privilege-based algorithm.

```
Senders (code of process $s_i$):
  Initialisation:
    $tosend_{s_i} \leftarrow \emptyset$
    if $s_i = s_1$ then
      send $token(seqnum : 1)$ to $s_1$
  procedure $TO\text{-}multicast(m)$                                   {To TO-multicast a message $m$}
    $tosend_{s_i} \leftarrow tosend_{s_i} \cup \{m\}$
  when receive $token$
    for each $m'$ in $tosend_{s_i}$ do
      send $(m', token.seqnum)$ to destinations
      $token.seqnum \leftarrow token.seqnum + 1$
    $tosend_{s_i} \leftarrow \emptyset$
    send $token$ to $s_{i+1}$

Destinations (code of process $p_i$):
  Initialisation:
    $nextdeliver_{p_i} \leftarrow 1$
    $pending_{p_i} \leftarrow \emptyset$
  when receive $(m, seqnum)$
    $pending_{p_i} \leftarrow pending_{p_i} \cup \{(m, seqnum)\}$
    while $\exists (m', seqnum') \in pending_{p_i}$ s.t. $seqnum' = nextdeliver_{p_i}$ do
      deliver$(m')$
      $nextdeliver_{p_i} \leftarrow nextdeliver_{p_i} + 1$
```

Observation 5
In synchronous systems, privilege-based algorithms are based on the idea that each sender process is allowed to send messages only during some predetermined time slots. These time slots are attributed to each process in such a way that no two process can send messages at the same time. By ensuring that the communication medium is accessed in mutual exclusion, the total order is easily guaranteed.

4.3.3 Moving Sequencer

Moving sequencer algorithms are based on the idea that a group of external processes successively act as sequencer. The responsibility of sequencing messages is passed among these processes. In Figure 4.6, the sequencer is chosen among several external processes. It is however important to understand that, with moving sequencer algorithms, the roles of external and destination processes are normally combined. Unlike privilege-based algorithms, the sequencer is related to the destinations rather than the senders.

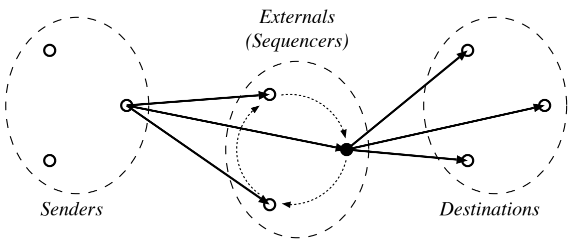


Figure 4.6: Moving sequencer algorithms.

Algorithm 4.3 illustrates the principle of moving sequencer algorithms. Messages receive a sequence number from a sequencer. The messages are then delivered by the destinations in increasing sequence numbers.

To multicast a message m, a sender sends m to the sequencers. Sequencers circulate a token message that carries a sequence number and a list of all messages for which a sequence number has been attributed (i.e., sequenced messages). Upon reception of the token, a sequencer assigns a sequence number to all received yet unsequenced messages. It sends the newly sequenced messages to the destinations, updates the token, and passes it to the next sequencer.

Observation 6
The major motivation for passing the token is to distribute the load of sequencing the messages among several processes.

Observation 7
It is tempting to consider that privilege-based and moving sequencer algorithms are equivalent. Indeed, both techniques rely on some token passing mechanism and hence use similar mechanisms. However, both techniques differ in one important aspect: the total order is built by senders in privilege-based algorithms, and by external processes (sequencers) in moving sequencer algorithms. This has at least two major

Algorithm 4.3 Simple moving sequencer algorithm.

```
Sender:
  procedure TO-multicast($m$)                                  {To TO-multicast a message $m$}
    send ($m$) to all sequencers

Sequencers (code of process $s_i$):
  Initialisation:
    $received_{s_i} \leftarrow \emptyset$
    if $s_i = s_1$ then
      send $token(sequenced : \emptyset, seqnum : 1)$ to $s_1$
  when receive $m$
    $received_{s_i} \leftarrow received_{s_i} \cup \{m\}$
  when receive $token$ from $s_{i-1}$
    for each $m'$ in $received_{s_i} \setminus token.sequenced$ do
      send $(m', token.seqnum)$ to destinations
      $token.seqnum \leftarrow token.seqnum + 1$
      $token.sequenced \leftarrow token.sequenced \cup \{m'\}$
    send $token$ to $s_{i+1}$

Destinations (code of process $p_i$):
  Initialisation:
    $nextdeliver_{p_i} \leftarrow 1$
    $pending_{p_i} \leftarrow \emptyset$
  when receive $(m, seqnum)$
    $pending_{p_i} \leftarrow pending_{p_i} \cup \{(m, seqnum)\}$
    while $\exists (m', seqnum') \in pending_{p_i}$ s.t. $seqnum' = nextdeliver_{p_i}$ do
      deliver $(m')$
      $nextdeliver_{p_i} \leftarrow nextdeliver_{p_i} + 1$
```

consequences. First, moving sequencer algorithms are easily adapted to open groups. Second, in privilege-based algorithms the passing of token is necessary to ensure the liveness of the algorithm. In contrast, the passing of token in moving sequencer algorithms is to ensure load balancing (and sometimes collect acknowledgements).

4.3.4 Fixed Sequencer

In a fixed sequencer algorithm, one process is elected as the sequencer and is responsible for ordering messages. Unlike moving sequencer algorithms, there is only one such process, and the responsibility is not normally transfered to another process (at least in the absence of failures). On Figure 4.7, the sequencer is depicted by the solid black circle.

Algorithm 4.4 illustrates the approach. One specific process takes the role of a sequencer and builds the total order. This algorithm does not tolerate a failure of the sequencer.

To multicast a message m, a sender sends m to the sequencer. Upon receiving m, the sequencer assigns it a sequence number and relays m with its sequence number to the destinations. The destinations then deliver messages in sequence thus according to some total order.

Observation 8

There exists a second variant to fixed sequencer algorithms. Unlike Algorithm 4.4,

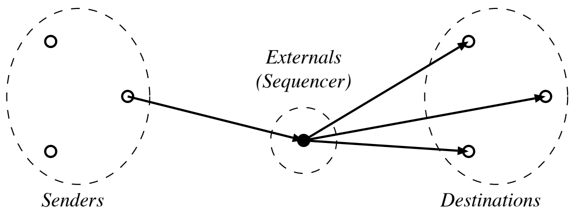


Figure 4.7: Fixed sequencer algorithms.

the sender sends its message m to the sequencer as well as the destinations. Upon receiving m, the sequencer sends a sequence number for m to all destinations. The destinations can then deliver messages in sequence after they have received both the message and its sequence number.

4.3.5 Destinations Agreement

In destinations agreement algorithms, the order is built by an agreement between the destination processes. The destinations receive messages without any ordering information, and exchange information in order to define an order. Messages are then delivered according to this order. This approach is illustrated on Figure 4.8 and by Algorithm 4.5.

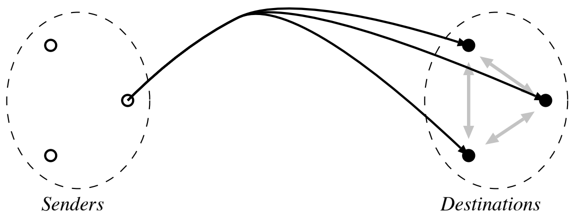


Figure 4.8: Destinations agreement algorithms.

To multicast a message m, a sender sends m to all destinations. Upon receiving m, a destination assigns it a local timestamp and multicasts this timestamp to all destinations. Once a destination process has received a local timestamp for m from all other destinations, a unique global timestamp $sn(m)$ is assigned to m as the maximum of all local timestamps. A message m can only be delivered once it has received its global timestamp $sn(m)$ and there is no other undelivered message m' can receive a smaller global timestamp $sn(m')$.

4.3.6 Discussion: Time-Free *vs.* Time-Based Ordering

We introduce a further distinction between algorithms, orthogonal to the ordering class. An important aspect of total order multicast algorithms is the use of physical time as a base for the ordering mechanisms. For instance, in Section 4.3.1 (see Alg. 4.1) we have presented a simple communication-history algorithm based on the use of *logical* time. It is indeed possible to design a similar algorithm based on the use of *physical* time and synchronized clocks instead.

Algorithm 4.4 Simple fixed sequencer algorithm (code of process p).

```
1: Sender:
2:   procedure TO-multicast($m$)                                  {To TO-multicast a message $m$}
3:     send ($m$) to sequencer

4: Sequencer:
5:   Initialisation:
6:     $seqnum \leftarrow 1$
7:   when receive ($m$)
8:     $sn(m) \leftarrow seqnum$
9:     send $(m, sn(m))$ to all
10:    $seqnum \leftarrow seqnum + 1$

11: Destinations (code of process $p_i$):
12:   Initialisation:
13:     $nextdeliver_{p_i} \leftarrow 1$
14:     $pending_{p_i} \leftarrow \emptyset$
15:   when receive $(m, seqnum)$
16:     $pending_{p_i} \leftarrow pending_{p_i} \cup \{(m, seqnum)\}$
17:     while $\exists(m', seqnum') \in pending_{p_i} : seqnum' = nextdeliver_{p_i}$ do
18:       deliver $(m')$
19:       $nextdeliver_{p_i} \leftarrow nextdeliver_{p_i} + 1$
```

Algorithm 4.5 Simple destinations agreement algorithm.

```
1: Sender:
2:   procedure TO-multicast($m$)                                  {To TO-multicast a message $m$}
3:     send ($m$) to destinations

4: Destinations (code of process $p_i$):
5:   Initialisation:
6:     $stamped_{p_i} \leftarrow \emptyset$
7:     $received_{p_i} \leftarrow \emptyset$
8:     $LC_{p_i} \leftarrow 0$                          {$LC_{p_i}$: logical clock of process $p_i$}
9:   when receive $m$
10:    $ts_i(m) \leftarrow LC_{p_i}$
11:    $received_{p_i} \leftarrow received_{p_i} \cup \{(m, ts_i(m))\}$
12:    send $(m, ts_i(m))$ to destinations
13:    $LC_{p_i} \leftarrow LC_{p_i} + 1$
14:  when received $(m, ts_j(m))$ from $p_j$
15:    $LC_{p_i} \leftarrow \max(ts_j, LC_{p_i} + 1)$
16:    if received $(m, ts(m))$ from all destinations then
17:      $sn(m) \leftarrow \max_{i=1 \cdots n} ts_i(m)$
18:      $stamped_{p_i} \leftarrow stamped_{p_i} \cup \{(m, sn(m))\}$
19:      $received_{p_i} \leftarrow received_{p_i} \setminus \{m\}$
20:      $deliverable \leftarrow \emptyset$
21:      for each $m'$ in $stamped_{p_i}$ such that $\forall m'' \in received_{p_i} : sn(m') < ts_i(m'')$ do
22:        $deliverable \leftarrow deliverable \cup \{(m', sn(m'))\}$
23:      deliver all messages in $deliverable$ in increasing order of $sn(m)$
24:      $stamped_{p_i} \leftarrow stamped_{p_i} \setminus deliverable$
```

In short, we discriminate between algorithms with time-free ordering and those with time-based ordering. Algorithms with *time-free ordering* are those which use an ordering mechanism that relies neither on physical time nor on the synchrony of the system for the ordering of messages. Conversely, algorithms with *time-based ordering* are those which do rely on physical time.

4.4 A Note on Asynchronous Total Order Broadcast Algorithms

In many papers written about Total Order Broadcast, the authors claim that their algorithm solves the problem of Total Order Broadcast in asynchronous systems with process failures. This claim is of course incorrect, or incomplete at best. Indeed, Total Order Broadcast is not solvable in the asynchronous system model as defined by Fischer, Lynch, and Paterson [FLP85]. Or put differently, it is not possible, in asynchronous systems with process failures, to guarantee both safety and liveness.

From a formal point-of-view, most practical systems are asynchronous because it is not possible to assume that there is an upper bound on communication delays. In spite of this, why do many practitioners still claim that their algorithm can solve Consensus related problems in real systems?

To answer this question, it is first of all important to understand that real systems usually exhibit some level synchrony, and are thus not exactly asynchronous. However, practitioners often rely on the rarely explicit assumption that "most messages are likely to reach their destination within a known delay δ" [CMA97, CF99]. Whether the bound δ is known or not does not actually matter. Indeed, the existence of such a *finite* bound means that the probabilistic behavior of the network is *known* and *stable*, thus increasing the strength of the model. In the sequel, we call such a system model an *ad-hoc asynchronous system model*. Hence, so-called asynchronous algorithms do not usually guarantee liveness in a purely asynchronous system model, but can only do so if the (implicit) assumptions of the ad-hoc asynchronous system model are met.

An ad-hoc asynchronous model can be related to a synchronous model with timing failures. Indeed, assuming that messages will meet a deadline $T+\delta$ with a given probability $P(T+\delta \text{ met})$ is equivalent to assuming that messages will miss the deadline $T+\delta$ (i.e., a timing failure) with a known probability $P(T+\delta \text{ missed}) = 1 - P(T+\delta \text{ met})$. This does not put a bound on the occurrence of timing failures, but puts a probabilistic restriction on the occurrence of such failures. At the same time, a purely asynchronous system model can also be seen as a synchronous system model with an unrestricted number of timing failures.

4.5 Survey of Existing Algorithms

We now consider the Total Order Multicast algorithms that are described in the literature. For each of the five classes of algorithms, we enumerate the algorithms that belong to that class, explain their respective differences and similarities, and discuss their behavior in the presence of failures. Although we discuss the various aspects of algorithms according to the previous sections of the chapter, we begin the discussion with the ordering mechanism, thus avoiding to remain too abstract.

4.5.1 Communication History Algorithms

Algorithms The following algorithms belong to the communication history class of algorithms and are further discussed in Section 4.5.1.

- Lamport [Lam78]
- Psync [PBS89]
- Total [MMSA93, MSMA90, MMSA91, MSM89, MMS99, MMS95]
- ToTo [DKM93]
- Newtop [EMS95]
- COReL [KD00, KD96]

Additionally, the following three algorithms rely on time-based ordering mechanisms.

- HAS atomic broadcast [CASD95, CDSA90, CASD84].
- Redundant broadcast channels [Cri90].
- Berman and Bharali [BB93].

Ordering mechanism Lamport [Lam78] uses his definition of logical clocks to define a total order on events. The principle of total order broadcast is then to decide that messages must be ordered according to the total order defined on their respective *send* events. Lamport describes a mutual exclusion algorithms that is based on a total order of events. Based on this mutual exclusion algorithm it is straightforward to derive a total order broadcast algorithm. Newtop [EMS95] is based on Lamport's principle but extends Lamport's algorithm in many ways (e.g., fault-tolerance, multiple groups; see discussions below).

Total [MMSA93, MSMA90, MMSA91, MSM89, MMS99, MMS95] is a total order broadcast algorithm that is build on top of a reliable broadcast algorithm called Trans (Trans in defined together with Total). Trans uses an acknowledgement mechanism that defines a partial order on messages. Total builds a total order that is an extension of this partial order.

ToTo [DKM93] is an "agreed multicast[5]" algorithm developed on top of the Transis partitionable group communication system [DM96]. ToTo extends the order of an underlying causal broadcast algorithm. It is based on dynamically building a causality graph of received messages. A particularity of Toto is that, to deliver a message m, a process must have received acknowledgements for m from as few as a majority of the processes (instead of all processes).

COReL [KD00] is a total order broadcast algorithm that sits on top of Transis. It aims at building consistent replicated services in partitionable systems like Transis. For this, it relies on

[5]Agreed Multicast [DKM93, ADKM92] is a problem based on Total Order Multicast, with a weaker definition to account for network partitions. Informally, an agreed multicast satisfies the properties of Total Order Multicast, as long as there is no partition. However, if partitions occur, then Agreement and Total Order are guaranteed within each partition, but are not necessarily satisfied across two different partitions.

an underlying "agreed multicast[5]" algorithm (e.g., ToTo). COReL gradually builds a global order by tagging messages according to three different color levels (red, yellow, green). A message starts at the red level, when a process has no knowledge about its position in the global order. As more information becomes available, the message is promoted to a higher level, until it turns green. The message can then be delivered as its position in the global order is then known.

Psync [PBS89] is a total order broadcast algorithm used in the context of two group communication systems: Consul [MPS93], and Coyote [BHSC98]. In Psync, processes dynamically build a causality graph of messages as they receive new messages. Psync then delivers messages according to a total order that is an extension of the causal order.

Communication history algorithms with time-based ordering use physical clocks instead of logical ones. Thus, the order of messages is defined by assigning a physical timestamp to *send* events. The algorithms rely on the assumption that (1) physical clocks are synchronized, and (2) there is a known upper bound on communication delays.

Cristian, Aghili, Strong, and Dolev [CASD95] propose a collection of total order broadcast algorithms (called HAS) that assume a synchronous system model with ϵ-synchronized clocks. The authors describe three algorithms—HAS-$\mathcal{O}$, HAS-$\mathcal{T}$, and HAS-$\mathcal{B}$—that are respectively tolerant to omission failures, timing failures, and authenticated Byzantine failures. These algorithms are based on the principle of *information diffusion*, which is itself based on the notion of flooding or gossiping. In short, when a process wants to broadcast a message m, it timestamps it with the time of emission T according to its local clock, and sends it to all neighbors. Whenever a process receives m for the first time, it relays it to its own neighbors. Processes deliver message m at time $T + \Delta$, according to their timestamp (where Δ is the termination time; a known constant that depends on the topology of the network, the number of failures tolerated, and the maximum clock drift ϵ).

Cristian [Cri90] presents an adaption of the HAS algorithm for omission failures (HAS-$\mathcal{O}$) to the context of broadcast channels. The system model assumes the availability of $f + 1$ independent broadcast channels (or networks) that connect all processes together, thus creating $f + 1$ independent communication paths between any two processes (where f is the maximum number of failures). As a result of this architecture, the algorithm can achieve a significant message reduction over HAS-$\mathcal{O}$.

Berman and Bharali [BB93] present a technique that allows to transform an early-stopping Byzantine Agreement algorithm into a total order broadcast algorithm with similar early-stopping properties.

System model Lamport's algorithm [Lam78] assumes a failure-free asynchronous system model (note that, due to the absence of failures, the FLP impossibility result does not apply).

ToTo [DKM93] relies on the Transis group membership service [DM96, Mal94, ADKM92]. Transis is a partitionable group membership and thus is not based on process controlled crash. Instead, the system allows partitioned groups to diverge.

COReL [KD00] relies on the weaker guarantees offered by algorithms such as ToTo, in addition to the group membership. Unlike ToTo, COReL does not allow disconnected groups to diverge, but relies for correctness (agreement, termination) on the assumption that disconnected processes eventually reconnect.

Total [MMSA93, MSMA90, MMSA91, MSM89] assumes an ad hoc asynchronous system model with process crash and message loss. If the system is fully asynchronous, the algorithm is only partially correct (i.e., safe but not live). Moser and Melliar-Smith [MMS99, MMS95] also describe an extension of Total to tolerate Byzantine failures.

Psync [PBS89] assumes an ad hoc asynchronous system model. As for failures, Psync assumes that process may crash (permanent failures) and that messages can be lost (transient failures).

As already mentioned, the HAS algorithms [CASD95] and the adaptation to redundant broadcast channels [Cri90], assume a synchronous model with ϵ-synchronized clocks. The HAS algorithms consist of three algorithms tolerant to omission, timing, and (authenticated) Byzantine failures. The adaptation to redundant broadcast channels is tolerant only to omission failures.

Berman and Bharali [BB93] give three methods to transform Byzantine Agreement into total order broadcast. These methods are respectively adapted to the round synchronous model, the synchronous model with ϵ-synchronized clocks, and the synchronous model with timing uncertainty as described by Attiya, Dwork, Lynch, and Stockmeier [ADLS94]. The resulting algorithms are tolerant to Byzantine failures.

Specification Most communication history algorithms are uniform, or can easily adapted to become uniform. In fact, a uniform communication history algorithm does not generate more messages than a non-uniform one. The difference is that a non-uniform algorithm can normally deliver messages earlier than a uniform one. Satisfying uniform properties is actually related to the information passed by messages, to the delivery condition, to the actual characteristics of the group membership (or reconfiguration protocol), and to the failure model. For instance, Byzantine algorithms cannot be uniform (see Sect. 4.1.2).

Communication history algorithms can only make progress when processes communicate often with each other. So, to ensure liveness, all of these algorithms require (sometimes implicitly) that processes send empty messages if they have not sent messages for a long period. In algorithms with time-based ordering, physical time can efficiently replace empty messages [Lam84].

According to the proofs, Newtop [EMS95] ensures only Agreement and Total Order. Although it is not proven, it seems that the algorithm actually ensures Uniform Agreement and Uniform Total Order.

Psync [PBS89] is left unspecified. From the description of the algorithm, it is not hard to believe that it actually builds a total order in the absence of failures. However, the exact properties enforced by the algorithm in the presence of failures are difficult to guess.

ToTo [DKM93] solve a problem called Agreed multicast that allows partitioned groups to diverge (and are proven correct for this). Thus, the algorithm does not solve the problem of total order broadcast if processes can be partitioned. Nevertheless, if we consider the algorithm in an environment where network partitions never occur, it probably satisfies Agreement and Total Order.

The problem solved by COReL [KD00] does not allow disconnected groups to diverge, but it must weaken the termination property: "[COReL] guarantees that if a majority of the processes form a connected component then these processes eventually deliver all messages sent by any of them, in the same order." It is not difficult to see that, assuming that all correct processes are always connected, COReL satisfies the problem of Total Order Multicast defined in Section 4.1. In that case, it is probably easy to adapt the proofs in order to show that COReL satisfies both Uniform Agreement and Uniform Total Order.

All time-free communication history algorithms extend the partial order defined by causality. As a consequence, every communication history algorithm delivers messages according to a causal total order.

Cristian et al. [CASD95] prove that the three HAS algorithms satisfy Agreement. The authors do not prove Total Order but, by the properties of synchronized clocks and the timestamps, Uniform Total Order is not too difficult to enforce. However, if the synchronous assumptions do not hold, the algorithms may probably violate Total Order rather than just Termination.

Berman and Bharali [BB93] show that the resulting Atomic Broadcast algorithm inherits the properties of the underlying Byzantine Agreement algorithm. However, since the authors are not concerned with uniformity, they only mention that the resulting Atomic Broadcast algorithm must satisfy Agreement and Total Order.

Architectural properties In communication history algorithms, the destination processes usually require information from all potential senders in order to determine when a message can be delivered. For instance, in Algorithm 4.1 the destinations can deliver a message m only after they know that they will only receive new messages with a larger timestamp. This implies that destinations have some knowledge of the senders, and thus makes the approach poorly suited to open groups.

Aside from Newtop [EMS95] none of the communication history algorithm presented in this chapter are designed to support multiple groups. To support multiple groups, Newtop uses a mechanism that is in essence similar to Skeen's algorithm, as described by Birman and Joseph [BJ87] (Skeen's algorithm and its variants are discussed in Section 4.5.5).

HAS [CASD95] and its extension to broadcast channels [Cri90], and Berman and Bharali [BB93] are not conceptually restricted to closed groups. However, the assumed upper bound on communication delays, the assumed network topology, and the synchronized clocks[6] put some practical restrictions on the openness of the architecture.

Failure management The solution proposed by Lamport [Lam78] does not consider failures. In fact, the crash of a process blocks the algorithm forever. Lamport's solution to total order can easily be made fault-tolerant by relying on a group membership. Newtop [EMS95] is a good example of this.

[6]With GPS-based clock synchronization systems, clock synchronization is probably not a strong constraint, though.

Trans/Total does not rely on a group membership, but rather on its own failure detection scheme. Trans is designed to run on a broadcast network and uses a combination of positive and negative acknowledgements piggybacked on actual messages.

ToTo relies on the Transis group membership service to tolerate failures. This algorithm needs the group membership service to ensure its liveness in the case of failures and partitions, but its safety does not rely on it.

COReL [KD00] tolerates process crashes and communication failures resulting in network partitions. The algorithm relies entirely on the underlying group membership (Transis) to tolerate failures. COReL requires that some protocol messages are logged in order to support the recovery of crashed processes.

Psync [PBS89] does not rely on a group membership to tolerate failures, but the actual mechanism is not described in details. The algorithm uses a negative acknowledgement to retransmit lost messages (thus implementing reliable channels). Crashed process are never excluded from a group. However, it seems that the crash of a process may lead some correct processes to *discard* messages while others don't, thus violating Agreement!

HAS [CASD95] tolerates omission, timing, or Byzantine failures. There is no exclusion in the algorithm, and the failures of processes and communication links are masked by redundancy. The algorithms must rely on the assumption that the subgraph consisting of all correct processes and all correct communication links is connected. In other words, the connectivity of the network is redundant enough that there is always a correct path between any two correct processes.

The adaptation to redundant broadcast channels [Cri90] tolerates communication failures and crash failures. The algorithm tolerates crash failures in the same way. However, communication failures are tolerated as follows: since the algorithm is based on the availability of $f + 1$ communication channels (where f is the maximum number of failures assumed), then there is at least one correct communication channel.

4.5.2 Privilege-Based Algorithms

Algorithms Six algorithms fall into the class of privilege-based algorithms, among which four rely on time-free ordering:

- TPM [RM89].
- Train protocol [Cri91].
- Totem [AMMS$^+$95, AMMS$^+$93].
- On-demand [CMA97, ACM95].

The following two algorithms rely on time-based ordering.

- Gopal and Toueg [GT89].
- MARS [KGR90].

Ordering mechanism In the absence of failures, almost all algorithms behave in a similar fashion to Algorithm 4.2 (p.60). With the exception of On-demand, all algorithms rely on the existence of a logical ring along which a token is passed. However, there are slight differences between those algorithms.

Although based on token-passing along a logical ring, the Train protocol [Cri91] is based on the idea that the token (called a train) carries the messages in addition to the privilege to broadcast. More specifically, when a sender receives the token, it adds the messages that it wants to broadcast at the end.

Unlike the other privilege-based algorithms, On-demand [CMA97] does not use a logical ring. Processes which want to broadcast a message must request the token by sending a message to the current token holder. As a consequence, the protocol is more efficient if senders send long bursts of messages and such bursts rarely overlap. Also, in contrast to the other algorithms, *all processes* must know the identity of the token holder at any time.

All the algorithms (except Train) restrict the token holding time, by putting a bound on the number of messages that a process can broadcast before passing on the token (in the case of time-based ordering, the amount of time that it can keep the token). On the one hand, this prevents starvation by a process too eager to send. On the other hand, this helps to avoid buffer overflows at the destinations, which otherwise would lead to expensive message retransmissions. In particular, TPM sets a fixed bound on the number of messages, whereas Totem and On-demand implement more sophisticated flow control strategies.

Finally, Gopal and Toueg's algorithm [GT89] and MARS [KGR90] are two algorithms that rely on physical time and synchronized clocks to order messages.

Gopal and Toueg's algorithm is based on the round synchronous model. During each round, one of the processes is designated as the *transmitter* (the only process allowed to broadcast messages). Messages are delivered once they are acknowledged, three rounds after their initial transmission.

The MARS algorithm takes a different approach. It is based on the principle of time-division multiple-access (TDMA). TDMA consists in attributing predefined time slots to each process. Processes are then allowed to broadcast messages during their time slots only. This can be seen as an example of using time to communicate [Lam84] (here, to pass the token).

System model The four algorithms with time-free ordering (TPM, Totem, Train, On-demand) assume an ad-hoc asynchronous system model with process crashes and message omission failures. Channels are lossy, but their exact semantics is never properly specified. All algorithms except Totem assume that at least a majority of the processes are always correct and never suspected (Totem requires the existence of only one such process). These algorithm are based on a group membership used to define the logical ring, and hence rely on process controlled crash to solve Total Order Broadcast.

The two algorithms with time based ordering (Gopal and Toueg, and MARS) assume a synchronous model with synchronized clocks. Gopal and Toueg is based on the round synchronous model. The algorithm assumes that crash failures as well as omission failures can occur. MARS use TDMA, which can actually be seen as some implementation of a round synchronous system. The MARS algorithm, as it is described by Kopetz et al. [KGR90], is only tolerant to crash failures.

Specification With the notable exception of Gopal and Toueg [GT89], none of the privilege-based algorithms studied here are formally specified. Nevertheless, it is possible to reason about those algorithms and infer some of their properties. In a failure-free environment, all time-free privilege-based algorithms mentioned above satisfy both Total Order and Agreement. It is however less clear what exactly happens to those properties if a failure occurs (especially if the token must be regenerated).

In these algorithms, the sender determines the total order by assigning a unique sequence number to each message. This means that the information from which the total order is inferred is carried by the messages themselves, and is directly interpretable by the destinations. Assuming a perfect failure detector, it is not difficult to ensure Uniform Total Order. As a second consequence, a message is always received with the same sequence number, hence Uniform Integrity only requires to keep track of the sequence number of the last message delivered.

According to their respective authors, the time-free algorithms are quite different with respect to Agreement. It is claimed that TPM and Train satisfy Uniform Agreement, while On-demand satisfies only Agreement. According to its authors, Totem leaves the choice to the user and provides both "agreed order" (non-uniform agreement) and "safe order" (uniform agreement).

No time-free privilege-based algorithms that satisfy Uniform Agreement allow contamination. Indeed, all processes deliver messages with consecutive sequence numbers, making it impossible to generate "holes" in the sequence of delivered messages.

Gopal and Toueg's [GT89] algorithm satisfies Uniform Total Order. However, it guarantees Uniform Agreement only if the transmitter is correct. The papers about MARS [KGR90, KDK$^+$89] do not provide enough information about the atomic multicast algorithm and its properties.

Architectural properties Due to their approach, all privilege-based algorithms are restricted to a single closed group. Most of the algorithms require that all sender processes are also destinations. Besides, as these algorithms introduce arbitration on the side of the senders, all senders must know each other. However, Rajagopalan and McKinley [RM89] include some ideas how to extend TPM to support multiple closed groups.

According to their designers, the ring and the token passing scheme make privilege-based algorithm highly efficient in broadcast LANs, but less suited to interconnected LANs. To overcome this problem, Amir et al. [AMMSB98] extend Totem to an environment consisting of multiple interconnected LANs. The resulting algorithm performs better in such an environment, but otherwise has the same properties as the original version (single ring).

Failure management All algorithms tolerate message loss by relying on message retransmission. The time-free algorithms (except On-demand) use the token to carry retransmission requests (i.e., negative acknowledgements). Rajagopalan and McKinley [RM89] also propose a variant of TPM in which retransmission requests are sent separately from the token, in order to improve the behavior in networks with a high rate of message loss. On-demand takes a different approach with positive acknowledgement for groups of messages that are sent directly by the destinations to the token holder.

Except for Gopal and Toueg's algorithm, all algorithms rely on a group membership service to exclude crashed (and suspected) processes. On top of that, the exclusion of suspected processes is unilaterally decided by only one of the processes. Except for On-demand, the detection of crashes is combined with the passing of the token.

Gopal and Toueg's algorithm considers differently the failure of the transmitter and that of other processes. First, the failure of a process different from the transmitter is simply ignored. If a process is unable to communicate with enough processes because of omissions, then it commits suicide (process controlled crash). Finally, according to Anceaume [Anc93b], the algorithm stops if the transmitter fails. This claim is however incomplete since Gopal and Toueg's algorithm leaves the system in such a state that the algorithm can safely be restarted.

4.5.3 Moving Sequencer Algorithms

Algorithms The following algorithms are based on a moving sequencer.

- Chang and Maxemchuck [CM84].
- RMP [WMK94].
- Pinwheel [CMA97, CM95].

To the best of our knowledge, there is no moving sequencer algorithm that relies on time-based ordering. It is questionable whether a time-based ordering would make sense in this context.

The three algorithms behave in a similar fashion. In fact, both Pinwheel and RMP are based on Chang and Maxemchuck's algorithm. They improve some aspects of Chang and Maxemchuck's algorithm, but in a different manner. Pinwheel is optimized for a uniform message arrival pattern, while RMP provides various levels of guarantees.

Ordering mechanism The three algorithms are based on the existence of a logical ring along which a token is passed. The process that holds the token, also known as the token site, is responsible for sequencing the messages that it receives. However, unlike privilege-based algorithms, the token site is not necessarily the sender of the message that it orders.

For practical reasons, all three algorithms require that the logical ring spans all destination processes. This requirement is however not necessary for ordering messages, and hence these algorithms still qualify as sequencer-based algorithms according to our classification.

System model All three algorithms assume an ad-hoc asynchronous system model with process crashes and message omission failures.

Pinwheel assumes that a majority of the processes remains correct and connected at all time (majority group). The algorithm is based on the timed asynchronous model [CF99]. It relies on physical clocks for timeouts, but it does not need to assume that these clocks are synchronized. The authors explicitly make the assumption that "most messages are *likely* to reach their destination within a known delay δ."

In order to deliver messages with strong requirements, RMP also assumes that a majority of the processes remain correct and connected at any time in order to satisfy the properties of Total Order Multicast.

Specification None of the three moving sequencer algorithms are formally specified. While it is easy to infer that all three algorithms probably satisfy Uniform Integrity, this is more difficult with the other two properties.

Chang and Maxemchuck's algorithm and Pinwheel seem to satisfy Uniform Total Order. In both algorithms, the uniformity is only possible through an adequate support from the group membership mechanism (or reformation phase).

Assuming an adequate definition of the "likeliness" of a message to reach its destination within a known delay, Chang and Maxemchuck seem to satisfy Uniform Agreement. Depending on the user's choice, RMP satisfies Agreement, Uniform Agreement, or neither properties. Pinwheel only satisfies Agreement, but Cristian, Mishra, and Alvarez argue that the algorithm could easily be modified to satisfy Uniform Agreement [CMA97]. This would only require that destination processes deliver a message after they have detected that it is stable.

Neither Chang and Maxemchuck nor Pinwheel are subject to contamination. RMP however does not preclude the contamination of the group.

Both Chang and Maxemchuck and Pinwheel deliver messages in a total order that is an extension of causal order. However, this requires the sender to be part of the destinations (i.e., closed group). It is also due to the constraint that a sender can broadcast a message only after the previous one has been acknowledged. As for RMP, this depends on the semantics that is required for each message.

Architectural properties Both Chang and Maxemchuck's algorithm and Pinwheel assume that the sender process is part of the logical ring, and hence one of the destinations (i.e., closed group). They use this particularity for the acknowledgement of messages. An adaptation to open groups would probably require only little changes, but then causal delivery would not be guaranteed.

RMP differs from the other two algorithms for it is designed to operate with open groups. Whetten, Montgomery, and Kaplan [WMK94] also claim that "RMP provides multiple multicast groups, as opposed to a single broadcast group." According to their description, supporting multiple multicast groups is a characteristic associated with the group membership service. It is then dubious that "multiple groups" is used with the same meaning as defined in Section 4.2.1, that is, total order is also guaranteed for processes that are at the intersection of two groups.

Failure management All three algorithms tolerate message loss by relying on a message retransmission protocol. The algorithms rely on a combination of negative and positive acknowledgements of messages. More precisely, when a process detects that it has missed a message, it issues a negative acknowledgement to the token site. On top of that, the token carries positive acknowledgements. The negative acknowledgement scheme is used for message retransmission, while the positive scheme is mostly used to detect the stability of messages.

4.5.4 Fixed Sequencer Algorithms

Algorithms The following algorithms are based on a fixed sequencer:

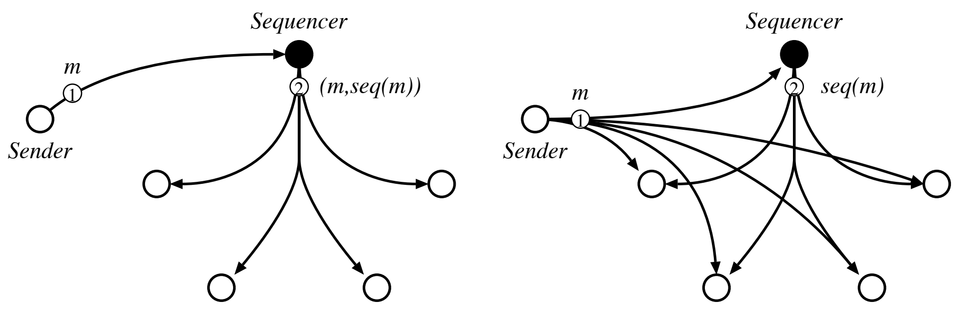
- Tandem global update [Car85], as described by Cristian et al. [CdBM94].
- Navaratnam, Chanson, and Neufeld [NCN88].
- Isis (sequencer) [BSS91].
- Amoeba (method 1, method 2) [KT91b, KT91a].
- Garcia-Molina and Spauster [GMS91, GMS89].
- Jia [Jia95], later corrected by Chiu and Hsiao [CH98], and Shieh and Ho [SH97].
- Phoenix [WS95].
- ATOP [CHD98].
- Newtop (asymmetric protocol) [EMS95].
- Rodrigues, Fonseca, and Veríssimo [RFV96].
- Rampart [Rei94b, Rei94a].

Ordering mechanism As briefly mentioned in Section 4.3.4, there are two basic variants to fixed sequencer algorithms. The first variant corresponds to Algorithm 4.4 (p.64) and is illustrated in Figure 4.9(a). In this case, when a process p wants to broadcast a message m, it only sends m to the sequencer. In turn, the sequencer appends a sequence number to m and relays it to all destinations. In the second variant (Fig. 4.9(b)), the process p sends its message m to all destination processes as well as the sequencer. The sequencer then broadcasts a sequence number for m, which is then used by the destinations to deliver m in the proper order.

The first variant (Fig.4.9(a)) is simple and generates very few messages. This approach is for instance taken by the Tandem global update protocol [Car85], the algorithm proposed by Navaratnam, Chanson, and Neufeld [NCN88], and Amoeba (method 1) [KT91b].

The second approach (Fig.4.9(b)) generates more messages, but it can reduce the load on the sequencer as well as make it easier to tolerate a crash of the sequencer. This second approach is taken by Isis (sequencer) [BSS91], Amoeba (method 2) [KT91b], Phoenix [WS95], and Rampart [Rei94b].

Newtop (asymmetric protocol) [EMS95], Garcia-Molina and Spauster [GMS91], and Jia's algorithm [Jia95] use more complicated techniques for ordering, as they cope with multiple overlapping groups. Here, the main difficulty is to ensure that two messages sent to different but overlapping groups are delivered in the same order on all the processes in the intersection of the



(a) Tandem-like variant

(b) Isis-like variant

Figure 4.9: Common variants of fixed sequencer algorithms.

two groups. When the system consists of a single destination group, the three algorithms are equivalent to the first variant illustrated in Figure 4.9(a).

In Newtop's asymmetric protocol, each group has an independent sequencer. All processes maintain logical clocks, and both the unicast from the sender to the sequencer and the multicast from the sequencer to the destination is timestamped. A process which belongs to multiple groups must delay the sending of a message (to the relevant sequencer) until it has received the sequence number for all previously sent messages (from the relevant sequencers).

A different approach is taken by Garcia-Molina and Spauster, as well as Jia. A propagation graph (a forest) is constructed first. Each group is assigned a starting node, and senders send their messages to these starting nodes. Messages travel along the edges of a *propagation graph*, and ordering decisions are resolved along the path. Jia [Jia95] creates simpler propagation graphs, using the notion of *meta-groups*, sets of processes having the same group membership (e.g., processes that belong simultaneously to a group G_1 and a group G_2, but to no other group, form a meta-group $G_{1\cap 2}$). If used in a single-group setting, these algorithms behave like Algorithm 4.4 (i.e., the propagation graph constructed in this case is a tree of depth 1).

Rodrigues, Fonseca, and Veríssimo [RFV96] present a special case, optimized for large networks. It uses a hybrid protocol for ordering messages: roughly speaking, on a local scale, ordering is decided by a fixed sequencer process, and on a global scale, the sequencer processes use a communication history based algorithm to decide on the order. More precisely, senders send their message to all destinations. Each sender has an associated sequencer process that issues a sequence number for the messages of the sender. Messages containing these sequence numbers are sent to all, and they are ordered using the communication history based algorithm. The authors also describe interesting heuristics to change a sequencer process to a non-sequencer process and vice versa. Reasons for such changes can be failures, membership changes or changes in the traffic pattern.

ATOP [CHD98] is also an "agreed multicast[7]" algorithm developed for the Transis group

[7]See footnote (5) on page 66.

communication system [DM96]. The originality of the algorithm is that it adapts to changes in the transmission rates, using some *adaptation policy* to influence the total order. The goal is to increase performance in systems where processes broadcast at very different rates.

System model All the algorithms assume an ad-hoc asynchronous system model, and rely on some form of process controlled crash: the killing of processes (e.g., Tandem), or the exclusion from a group membership (e.g., Isis, Rampart). Most of them only consider process crashes; Rampart is the only one of this class which tolerates Byzantine failures.

Tandem, Amoeba, and Garcia-Molina and Spauster's algorithm explicitly treat message omissions, whereas all the others rely on reliable channels. In addition, Isis (sequencer), Garcia-Molina and Spauster, Jia, and Rodriguez et al. require FIFO channels. Rampart not only requires FIFO channels, but also that all messages are authenticated.

Navaratnam, Chanson, and Neufeld [NCN88] consider a slightly more complicated failure model in which multiple processes may reside on the same machine and hence can fail in a dependent manner. However, this model is an easy extension of the usual model of independent process crashes.

Garcia-Molina and Spauster [GMS91] require synchronized clocks. However, synchronized clocks are only necessary to yield bounds on the behavior of the algorithm when crash failures occur. Neither the ordering mechanism, nor the fault tolerance mechanism needs them.

Specification Garcia-Molina and Spauster, Phoenix and Rampart explicitly specify whether the Agreement and the Total Order properties guaranteed by the algorithm are uniform or not.

Garcia-Molina and Spauster, as well as the extension proposed by Jia [Jia95] guarantee Uniform Total Order and Uniform Agreement (this is called reliability type R3 by Garcia-Molina and Spauster [GMS91]).

Phoenix consists of three algorithms with different guarantees. The first algorithm (weak order) only guarantees Total Order and Agreement. The second algorithm (strong order) guarantees both Uniform Total Order and Uniform Agreement. Then, the third algorithm (hybrid order) combines both semantics on a per message basis.

Because it considers Byzantine processes in its system model, Rampart only satisfies Agreement and Total Order.

The other algorithms are not specified so precisely, but it is still possible to reason about their properties. Actually, all of them satisfy only Total Order (non-uniform) and Agreement (non-uniform), except for Navaratnam, Chanson, and Neufeld for which this is unclear.

Isis (sequencer) and the closed group version of Jia[8] [Jia95, CH98] ensure causal order delivery in addition to total order. Tandem, Garcia-Molina and Spauster, and Jia's original algorithm provide FIFO order. Note that the latter two provide FIFO order because they assume FIFO reliable channels and any multicast starts with a unicast to a fixed process.

[8]The algorithm described by Jia [Jia95] violates causal order, but Chiu and Hsiao [CH98] provide a solution that only works in the case of closed groups.

Architectural properties As already mentioned when describing the ordering mechanisms, almost all of the algorithms are restricted to a single group, with the exception of Newtop (asymmetric protocol), Garcia-Molina and Spauster, and Jia.

Newtop (asymmetric protocol), Garcia-Molina and Spauster, and Jia's algorithm can manage multiple groups. In the two latter algorithm, the ordering of a message m may involve processes that are neither a destination nor the sender of m. This means that they only satisfy the Weak Minimality condition defined in Section 4.2.1.

We now discuss which algorithms have open groups or can be easily extended such that they have open groups. Some of the fixed sequencer algorithms require that the sender unicasts to the sequencer (see Fig.4.9(a), p.76). This requires that senders know which process is the sequencer *at any time*, i.e., they have to follow changes in group membership. Only group members have this information, therefore the algorithms in question—Tandem; Navaratnam, Chanson, and Neufeld; Amoeba (method 1)—are closed group algorithms. Extending these algorithms to open groups requires that all possible clients are notified of every membership change. Furthermore, clients must be ready to retransmit their requests if the sequencer crashes. As a result, this considerably limits the extent of such an adaptation.

The three algorithms which can manage multiple groups—Newtop (asymmetric protocol), Garcia-Molina and Spauster, and Jia—are open in the sense that a process in any of the groups they manage can send to any of these groups, not only to their own group. There is however a limitation to this: processes that are not managed by the algorithm cannot send messages to any of the groups. Also, Jia's algorithm satisfies causal order only if it is limited to operate with closed groups.

Most algorithms in which a sender starts by broadcasting to all members of the group (see Fig.4.9(b), p.76) can be adapted to open groups. Indeed, senders do not need to know the exact group membership if sending messages to excluded processes has no effect. Amoeba (method 2), Phoenix, and Rampart can be adapted in such a way. Also Rodrigues, Fonseca, and Veríssimo [RFV96] can be adapted to open groups under certain circumstances.[9] As for Isis (sequencer), the algorithm can be adapted to open groups, thus allowing processes to broadcast messages to a group to which they do not belong. In this case, total order broadcast could no more satisfy causal order.

Failure management All algorithms assume reliable channels except for Tandem, Amoeba, and Garcia-Molina and Spauster, which explicitly consider message omissions. Tandem uses positive acknowledgements for each message that comes from the sender. Garcia-Molina and Spauster use negative acknowledgements combined with live messages, if actual multicast messages are rare.[10] This ensures that processes can detect message losses fast. Amoeba uses a combination of positive and negative acknowledgements.[11]

[9]The algorithm can be adapted provided that one of the sequencers assumes responsibility for ordering messages issued by senders that are outside the group.

[10]The sequencer sends null messages to communicate the highest sequence number to the recipients.

[11]The actual protocol is complicated because it is combined with flow control, and tries to minimize the communication cost.

Almost all the algorithms use a group membership service to exclude faulty processes. Isis (sequencer), Phoenix, and Rodrigues, Fonseca, and Veríssimo [RFV96] do not specify this service in detail. In the group membership used by both Amoeba, and Navaratnam, Chanson, and Neufeld [NCN88], the exclusion of suspected processes is unilaterally decided by only one of the processes. In contrast, the membership service of Newtop requires that *all* processes that are not suspected suspect a process in order to exclude it from the group. Rampart needs that at least one third of all processes in the current view reach an agreement on the exclusion.[12] Considering that Rampart assumes that less than one third of the processes are Byzantine, this means that a single honest process (i.e., non-Byzantine) may unilaterally exclude another process from the group.

Garcia-Molina and Spauster; and Jia behave in a particular manner. If a non-leaf process p fails, then its descendants do not receive any message until p recovers. Hence, these two algorithms tolerate failures only in a model where every crashed process eventually recovers. Jia [Jia95] also suggests a limited form of group membership where processes from a pool of spare processes replace failed nodes in the propagation graph (inspired from Ng [Ng91]).

4.5.5 Destinations Agreement Algorithms

Algorithms The following algorithms fall into the class of destination agreement algorithms. They are further discussed in Section 4.5.5.

- Skeen (Isis) [BJ87].
- MTO [GS97a], corrected by SCALATOM [RGS98].
- TOMP [Das92].
- Fritzke, Ingels, Mostéfaoui, and Raynal [FIMR98]
- Luan and Gligor [LG90].
- Le Lann and Bres [LLB91].
- Chandra and Toueg [CT96].
- Prefix agreement [Anc97].
- Optimistic atomic broadcast [PS98].
- Generic broadcast [PS99].
- ABP [MA91b, Anc93a].
- ATR [DGF99].
- AMp [VRB89] and xAMp [RV92].

[12] This condition is of course necessary to prevent Byzantine processes from purposely excluding correct processes from the group.

Ordering mechanism In destinations agreement algorithms, the order is built solely by the destination processes. The destinations receive messages without any ordering information, and they exchange information in order to agree on the order.[13]

Destinations agreement algorithms are a little more difficult to relate to each other than algorithms of other classes. It is nevertheless possible to distinguish between two different families of algorithms. Algorithms of the first family use logical clocks similar to Algorithm 4.5 (p.64). In the second family, the algorithms are built around an agreement problem (e.g., Consensus, Atomic Commitment).

Algorithms of the first family are based on logical clocks, and generally operate according to the following steps:

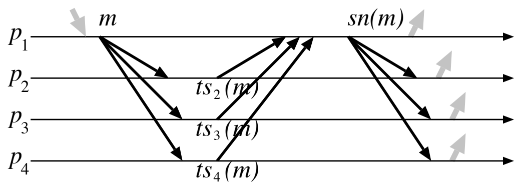
1. To multicast a message m, a sender sends m to all destinations.[14] Upon receiving m, a destination uses its logical clock to timestamp the message.

2. Destination processes communicate the timestamp of the message to the system, and a global timestamp $sn(m)$ is computed that is then sent back to all destinations.

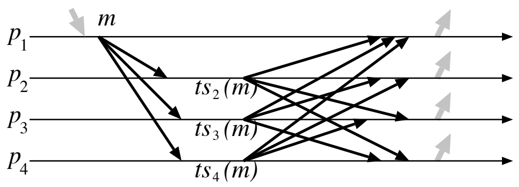
3. A message becomes deliverable once it has a global timestamp $sn(m)$. Deliverable messages are then delivered in the order defined by their global timestamp, provided that there is no other undelivered message m' that could possibly receive a smaller global timestamp $sn(m')$.

Skeen, TOMP, MTO/SCALATOM, and Fritzke et al. fall into the first family of destinations agreement algorithms.

In Skeen's algorithm, the sender coordinates the computation of the global timestamp. Destination processes send the local timestamp to the sender, which collects them. The sender then sends back the global timestamp once it is computed. Algorithm 4.5 is in fact largely inspired from Skeen's algorithm. The main difference is that Skeen's algorithm computes the global timestamp in a centralized manner, while Algorithm 4.5 does it in a decentralized way (see Fig.4.10).



(a) Skeen (centralized)



(b) Algorithm 4.5 (decentralized)

Figure 4.10: Centralized and decentralized approaches to compute a global timestamp

[13]The exchange of information may in fact involve the sender or some external process. However, these processes do not provide any ordering information, and merely arbitrate the exchange.

[14]Some algorithms require a Reliable Multicast to the destinations, depending on the fault tolerance properties of the algorithm.

MTO and SCALATOM are genuine multicast algorithms that are based on Skeen's algorithm. The global timestamp is computed using a variant of Chandra and Toueg's Consensus algorithm [CT96]. MTO requires that all messages are delivered in causal order. SCALATOM fixes some problems found in MTO and lifts the restriction on causal order by the introduction of an additional communication step.

Fritzke et al. also describe a genuine multicast algorithm. The algorithm uses a consensus algorithm to decide on the timestamp of a multicast message within each destination group (just like the algorithms following the second approach; see below). Then the destination groups exchange information to compute the final timestamp. Finally, a second consensus is executed in each group to update the logical clock.

TOMP is merely an optimization of Skeen's algorithm, based on piggybacking additional information on messages in order to deliver messages earlier. All comments made on Skeen's algorithm (as it appears in Isis) also apply to TOMP. For this reason, we stop to mention it in the sequel.

In the algorithms of the second family, processes agree on the message (or sequence of messages) to be delivered next, rather than on some timestamp. This agreement is usually obtained using either a Consensus or some Atomic Commitment protocol. Among the algorithms that belong to this family, Chandra and Toueg [CT96], Anceaume's prefix agreement [Anc97], optimistic atomic broadcast [PS98], generic broadcast [PS99] are based on Consensus.

In Chandra and Toueg's algorithm a sequence of Consensus is used to repeatedly agree on a set of received yet unordered messages. The order is then inferred from the set of messages, by the application of a predefined deterministic function.

Anceaume defines a variant of Consensus, called *Prefix Agreement*, in which processes agree on a stream of values rather than on a single value. Considering streams rather than single values makes the algorithm particularly well suited to the problem of Total Order Broadcast. The total order broadcast algorithm then repeatedly decides on the sequence of messages to be delivered next.

Optimistic atomic broadcast is based on the observation that messages that are broadcasted in a LAN are usually received in the same order by every process. Although the algorithm is still correct without this assumption, its performance are much better if it holds.

Generic Broadcast is not a total order broadcast per se. Instead, the order to enforce is a parameter of the algorithm. Depending on this parameter, the ordering properties of generic broadcast can vary from a simple unordered broadcast to total order. The strong point of this algorithm is that performance varies according to the required "amount of ordering".

Still in the second family of algorithms, Luan and Gligor [LG90], ATR [DGF99], Minet and Anceaume's ABP [MA91a], AMp [VRB89], xAMp [RV92], and Le Lann and Bres [LLB91] are based on some voting protocol, inspired from two and three phase commit.

In AMp and xAMp, the algorithm relies on the assumption that messages are most of the time received by all destinations in the same order (not unrealistic in LANs). So, the sender initiates some commit protocol when its sends its message. If the message is received by all destination processes in order, then the outcome is positive and all destinations commit and hence deliver the message. Otherwise, the message is rejected and the sender must try again.

In a nutshell, ABP and Luan and Gligor's algorithm are similar to each other. In both algorithms, the order of message is pre-established in an optimistic manner (assuming that no failure occurs). Then, the order is validated with a commit protocol before message can actually be delivered.

Delporte and Fauconier [DGF99] describe their total order algorithm on top of a layer which provides the abstraction of synchronized rounds. A multicast takes two rounds: in even rounds, all processes send an ordered set of messages to each other, and thus construct the order. In the subsequent round (odd), the order is validated, and messages can be delivered.

System model Most of the destinations agreement algorithms assume an asynchronous system model with failure detectors, reliable channels, and process crashes. For instance, Chandra and Toueg [CT96], Prefix Agreement [Anc97], MTO/SCALATOM [GS97c, RGS98], Fritzke et al. [FIMR98], optimistic atomic broadcast [PS98] require a $\Diamond\mathcal{S}$ failure detector, and assume that a majority of the processes are correct. Generic broadcast [PS99] also needs a $\Diamond\mathcal{S}$ failure detector, but it presupposes that two thirds of the processes are correct. Delporte and Fauconier's ATR algorithm [DGF99] relies on a $\Diamond\mathcal{P}$ failure detector and reliable FIFO channels. The authors argue that their algorithm enforces additional guarantees if the system is synchronous.

Other destinations agreement algorithms assume an ad hoc asynchronous system model, with process crashes and lossy channels.[15] These algorithms are Skeen's algorithm (Isis) [BJ87], Luan and Gligor [LG90], and Le Lann and Bres [LLB91]. The articles do not specify the exact conditions under which the algorithms are live; the assumption of an ad hoc asynchronous system model is implicit. Luan and Gligor's algorithm requires FIFO channels and that a majority of processes are correct.[16] Le Lann and Bres require an upper bound on the total number of failures (crash and omission). Skeen's algorithm (Isis) is based on a group membership service (Isis). The group membership service is needed for the liveness of the algorithm; without it, the algorithm would block with the crash of a single process.

Minet and Anceaume's ABP [MA91a], AMp and xAMp [VRB89, RV92] assume a synchronous system model. These algorithms rely on a group membership service, and assume that there is a known bound on the number of omission failures that a message can suffer.

Specification It is not difficult to see that all destinations agreement algorithms satisfy Uniform Integrity. Most of them also satisfy Uniform Agreement and Total Order. The articles in which the following algorithms are described prove the two latter properties (or at least sketch the proofs): Chandra and Toueg's algorithm [CT96], MTO/SCALATOM [GS97c, RGS98], Anceaume's prefix agreement [Anc97], and ATR [DGF99].

[15]The exact semantics of the lossy channels is rarely specified.

[16]In fact, Luan and Gligor's algorithm requires a little more than a majority of correct processes, but we do not go into such details.

The authors of Optimistic atomic broadcast [PS98] and Generic broadcast [PS99] only prove Agreement and Total Order. Similarly, Fritzke et al. [FIMR98] only prove Agreement and Total Order. Nevertheless, those algorithms probably also ensure Uniform Agreement and Uniform Total Order.

The following algorithms also probably satisfy the uniform properties: Luan and Gligor's algorithm [LG90] (unclear for Uniform Agreement), Le Lann and Bres [LLB91], ABP [MA91a], AMp and xAMp [VRB89, RV92] (unclear for Uniform Total Order).

Skeen's algorithm [BJ87] only guarantees Total Order. Depending on the group membership service, Skeen's algorithm can possibly guarantee Uniform Agreement.

Few articles are concerned with the problem of contamination. Nevertheless, Fritzke et al., and Minet and Anceaume (ABP) explicitly state that their algorithm prevents contamination. This is probably also true for the other algorithms.

Architectural properties The protocols which use logical clock values (Isis/Skeen, TOMP, MTO/SCALATOM, and Fritzke et al.) provide multiple groups ordering with the Strong Minimality property [GS97c] (see Sect. 4.2.1): only the sender and the destinations are involved in a multicast. The other algorithms only provide single group ordering.

The very structure of destinations agreement algorithms makes them relatively easy to adapt to open groups. Indeed, even if the sender is sometimes involved for coordination, it does not contribute to the ordering as such.

Skeen (Isis), Luan and Gligor, and ABP are closed group algorithms for the following reasons: the sender coordinates ordering, and the mechanism that handles failures requires the sender to be a member of the group. By transferring the coordination to one of the destinations, the algorithms could be adapted to open groups.

ATR only supports closed groups because of the properties of the layer it is built on that provides the abstraction of synchronized rounds. Also, AMp and xAMp quite heavily rely on the assumption that the network transmits messages in a certain order, and liveness can be violated is this does not hold often enough. In addition, this assumption may not be valid in an environment of interconnected subnetworks. As this puts a restriction on the processes which can broadcast messages, it is difficult to adapt the algorithm to open groups.

Failure management Skeen (Isis) [BJ87], Luan and Gligor [LG90], Le Lann and Bres [LLB91], ABP [MA91a], and AMp/xAMp [VRB89, RV92] explicitly tolerate message losses by retransmissions. Skeen uses a windowed point-to-point acknowledgement protocol. Note that this is nothing else but an explicit implementation of reliable channels. In the voting scheme of ABP, AMp and xAMp, the sender waits for all replies, thus it is natural to use positive acknowledgements. In Luan and Gligor's algorithm [LG90], the sender only waits for a majority of replies, thus the use of negative acknowledgements is more natural.

All of the algorithms tolerate process crashes. Skeen (Isis), ABP, AMp and xAMp rely on a group membership service and thus process controlled crash for failure management. Without a group membership service, these algorithms would be blocked even by a single failure. Unlike the others, the group membership service used by ABP requires the agreement of a majority of the processes to exclude a process (in the other algorithms, unilateral exclusion is possible).

There is a consensus algorithm (or a variant) based on the rotating coordinator paradigm at the heart of most other algorithms. In short, the rotating coordinator paradigm goes as follows: if the current coordinator is suspected, another process takes its place and tries to finish the algorithm; the suspected coordinator continues to participate in the algorithm as a normal process. More specifically, destinations agreement algorithms that are based on this mechanism are Chandra and Toueg [CT96], MTO/SCALATOM [VRB89, RV92], Fritzke et al. [FIMR98], Prefix agreement [Anc97], Optimistic atomic broadcast [PS98], and Generic broadcast [PS99].

In Luan and Gligor's algorithm [LG90], the voting scheme does not block if a process crashes, and there are mechanisms to bring slow replicas[17] up to date. If a sender fails, a termination protocol decides on the outcome of the multicast (commit or abort).

In ATR [DGF99], the layer that provides the abstraction of synchronized rounds also takes care of handling process crashes. *Phases* roughly correspond to the views of a group membership service, but provide less guarantees and are supposedly easier to implement.

4.6 Other Work on Total Order and Related Issues

Hardware-based protocols Due to their specificity, we have deliberately excluded all algorithms that make explicit use of dedicated hardware. Hence, they deserve to be cited as related work. Hardware-based protocols rely on specific hardware systems in various ways and for different reasons. Some protocols are based on a modification of the network controllers [CFM87, DKV83, Jal98, MA91a]. The goal of these protocols is to slightly modify the network so that it can be used as a virtual sequencer. In our classification system, these protocols can be assimilated as fixed sequencer protocols. Some other protocols rely on the characteristics of specific networks such as a specific topology [CL96] or the ability to reserve buffers [CMMS96]. Finally, some protocols are based on a specific network architecture and require that some computers are used for the sole purpose of serving the protocol [FM90, Tse89]. These protocols are designed with a specific application field in mind, and hence sacrifice flexibility for the sake of performance or scalability issues.

Formal methods Formal methods have been also applied to the problem of total order broadcast, in order to prove or verify the ordering properties of broadcast algorithms [ZH95, TFC99]. This work is however beyond the scope of this dissertation.

Group communication *vs.* transaction systems A few years ago, Cheriton and Skeen [CS93] began a polemic about group communication systems that provide causally and totally ordered communication. Their major argument against group communication systems was that systems based on transactions are more efficient while providing a stronger consistency model. This was

[17]A slow replica is one that always fails to reply on time.

later answered by Birman [Bir94] and Shrivastava [Shr94]. Almost a decade later it seems that this antagonistic view was a little shortsighted considering that the work on transaction systems and on group communication systems tend to influence each other for a mutual benefit [WPS99, PGS98, AAEAS97, SR96, GS95a, GS95b].

4.7 Summary

In spite of the sheer number of Total Order Multicast algorithms that have been published, most of them are merely improvements of existing ones. As a result, there are actually few algorithms as such, but a large collection of various optimizations. Nevertheless, it is important to stress that clever optimizations of existing algorithms often have an important effect on performance. For instance, Friedman and Van Renesse [FVR97] show that piggybacking messages can significantly improve the performance of algorithms.

Even though the specification of Atomic Broadcast dates back to some of the earliest publications about the subject (e.g., [CASD85]), few papers actually specify the problem they solve. In fact, too few algorithms are properly specified, let alone proven correct. Luckily, this is changing as the authors of many recent publications actually specify and prove their algorithms (e.g., [DKM93, CT96, Anc97, RGS98, PS98, DGF99]). Without pushing formalism to extremes, a clear specification and sound proofs are as important as the algorithm itself. Indeed, they clearly define the limits within which the algorithm can be used.

Chapter 5

Metrics for Asynchronous Distributed Algorithms

A scientific truth does not triumph by convincing its opponents
and making them see the light,
but rather because its opponents eventually die
and a new generation grows up that is familiar with it.

— **Maximilian Ernst Ludwig Planck** (1858–1947)

Performance prediction and evaluation are a central part of every scientific and engineering activity including the construction of distributed applications. Engineers of distributed systems rely heavily on various performance evaluation techniques and have developed the techniques necessary for this activity. In contrast, algorithm designers invest considerable effort in proving the correctness of their algorithms (which they of course should do!), but often oversee the importance of predicting the performance of those algorithms, i.e., they rely on simplistic metrics. As a result, there is a serious gap between the prediction and the evaluation of performance of distributed algorithms.

Performance prediction *vs.* evaluation of algorithms When analyzing performance, one has to make a distinction between their prediction and their evaluation. Performance prediction gives an indication of the expected performance of an algorithm, *before* it is actually implemented. Performance prediction techniques give fairly general yet imprecise information, and rely on the use of various metrics. Conversely, performance evaluation is an *a posteriori* analysis of an algorithm, once it has been implemented and run in a given environment (possibly a simulation). While the information obtained is usually very accurate and precise, the results depend on specific characteristics of the environment and thus lack generality. Performance prediction and evaluation are complementary techniques. Performance prediction is used to orient design decisions, while performance evaluation can confirm those decisions and allows the dimensioning of the various system parameters.

Definition of a metric This chapter investigates the problem of predicting and comparing the performance of distributed algorithms. The goal is to investigate metrics that answer typical questions such as choosing the best algorithm for a particular problem, or identifying the various performance tradeoffs related to the problem. A *metric* is a value associated with an algorithm, that has no physical reality and is used to define an order relation between algorithms. A good metric should provide a good approximation of the performance of an algorithm, regardless of the implementation environment. Even though some performance evaluation techniques are also based on an abstract model of the system (e.g., analytical approaches, simulation), a metric must be an easily *computable* value. This is clearly in contrast with simulation techniques that can model details of the system and the environment, thus use complex models and rely on a probabilistic approach.

Existing metrics for distributed algorithms Performance prediction of distributed algorithms is usually based on two rather simplistic metrics: time and message complexity. These metrics are indeed useful, but there is still a large gap between the accuracy of the information they provide, and results obtained with more environment specific approaches.

Time complexity measures the latency of an algorithm. There exist many definitions that are more or less equivalent. A common way to measure the time complexity of an algorithm (e.g., [ACT98, Sch97, Lyn96, Kri96, HT93, Ray88]) consists in considering the algorithm in a model where the message delay has a known upper bound δ. The efficiency of the algorithm is measured as the maximum time needed by the algorithm to terminate. This efficiency is expressed as a function of δ, and is sometimes called the *latency degree* [Sch97]. Time complexity is a *latency-oriented* metric because it measures the cost of *one* execution of the algorithm, from start to termination.

Message complexity is an estimation for the throughput of an algorithm. It consists in counting the total number of messages generated by the algorithm [Lyn96, HT93, ACT98]. This metric is useful when combined with time complexity, since two algorithms that have the same time complexity can generate a different volume of messages. Knowing the number of messages generated by an algorithm gives a good indication of its scalability and the amount of resources it uses. Furthermore, an algorithm that generates a large number of messages is likely to generate a high level of network contention.

Resource contention Resource contention is often a limiting factor for the performance of distributed algorithms. In a distributed system, the key resources are (1) the CPUs, and (2) the network, any of which is a potential bottleneck. The major weakness of the time and message complexity metrics is that neither gives enough importance to the problem of resource contention. Message complexity somehow takes account of network contention, but it ignores the contention on the CPUs. Time complexity ignores the problem of contention completely.

Contribution and structure In this chapter, we present a set of metrics to quantify both the latency and the throughput of distributed algorithms [UDS00a]. These metrics take account of both network and CPU contention, thus giving more accurate results than the conventional complexity metrics. Chapter 6 then provides a large example of the use of these metrics.

The rest of this chapter is structured as follows. First, Section 5.1 briefly surveys existing work on measuring distributed algorithms, parallel algorithms, and resource contention. Section 5.2 defines the system model on which the metrics are based. Finally, Section 5.3 presents the metrics per se.

5.1 Related Work

Time complexity Hadzilacos and Toueg [HT93] measure the time complexity of an algorithm by studying the algorithm in the context of the *synchronous round* model. The efficiency of the algorithm is then measured in terms of the number of rounds required for termination.

In order to compare algorithms that solve consensus, Schiper [Sch97] introduced the concept of *latency degree*. The latency degree of an algorithm is defined as the minimal number of communication steps required by an algorithm to solve a given problem. Given the assumption that no failures (crash or timing) occur, the latency degree gives a lower bound on the execution of the algorithm.

A similar metric has been used by Aguilera, Chen, and Toueg [ACT98] to express the time complexity of their consensus algorithm in the crash-recover model. With this metric, they consider their algorithm in a model where the message delay is bound by some known δ, including the message processing time. The efficiency of the algorithm is expressed as the maximum time needed by the algorithm to terminate, in the best possible run (no failure of any type). This efficiency is expressed as a factor of δ, where the factor is nothing but the latency degree.

In her book, Lynch [Lyn96] gives a more formal definition for the time complexity of distributed algorithms. Similar to the measure used by Aguilera, Chen, and Toueg [ACT98], she proposes to consider an upper bound d on the transmission delay. In addition, she also considers the execution time of tasks.

These metrics give similar information on the efficiency of a given algorithm. In the remainder of this paper, we will only consider the latency degree since all measures of time complexity give similar information. It is important to note here that none of these metrics take into account the message contention generated by the algorithm itself.

Message complexity The message complexity of a distributed algorithm is usually measured by the number of messages generated by a typical run of the algorithm (e.g., [Lyn96, HT93, ACT98]). This metric is useful in combination with time complexity, since two algorithms that have the same time complexity can generate a different volume of messages. Knowing the number of messages generated by an algorithm gives a good indication on its potential scalability. Furthermore, an algorithm that generates a large number of messages is likely to have a high level of network contention.

Resource contention in network models Resource contention (also sometimes called congestion) has been extensively studied in the literature about networking. The bulk of the publications about resource contention describe strategies to either avoid or reduce resource contention (e.g. [HP97, HYF99]). Some of this work analyze the performance of the proposed strategies. However, these analyses consist in performance *evaluation* and use models that are often specific to a particular network (e.g., [LYZ91]). Furthermore, the models cannot be adapted to our problem since they model distributed systems at the transport layer or below, whereas distributed algorithms are generally designed on top of it.

Resource contention in parallel systems Dwork, Herlihy, and Waarts [DHW97] propose a complexity model for shared-memory multiprocessors that specifically takes contention into account. This model is very interesting in the context of shared memory systems but is not well suited to the message passing model that we consider here. The main problem is that the shared memory model is a high-level abstraction for communication between processes. As such, it hides many aspects of communication that are important in distributed systems. Dwork, Herlihy, and Waarts associate a unit cost based on the access to shared variables, which has a granularity that is too coarse for our needs.

Computation models for parallel algorithms Unlike distributed algorithms, many efforts have been pursued to develop performance prediction tools for parallel algorithms. However, these models are poorly adapted to the execution of asynchronous distributed algorithms.

The family of PRAM models (e.g., [Kro96]) is defined as a collection of processors that have access to a global shared memory and execute the same program in lockstep. Variants of PRAM models are defined to restrict concurrent memory access, thus modeling resource contention. However, the computational model assumes that processors are tightly coupled and behaves synchronously (e.g., lockstep execution). On top of that, it is difficult to adapt to message passing (e.g., global shared memory, concurrent accesses to different memory locations).

To reduce the synchrony of the PRAM model, Gibbons, Matias, and Ramachandran [GMR98] define the QRQW asynchronous PRAM model, that does not require processors to evolve in locksteps, and is thus better adapted for asynchronous parallel algorithms. Although this model partly accounts for contention, the communication between processes is still based on a shared memory model. The computational model is hence unadapted to distributed algorithms.

Valiant [Val90] defines the BSP model (bulk synchronous parallel model) in which processors communicate by exchanging messages. However, the cost of communication is partly hidden. Furthermore, the communication occurs through periodic global synchronization operations that enforces a *round synchronous* computational model, as for instance described by Lynch [Lyn96].

The LogP model [CKP$^+$96] also considers that processors communicate by exchanging messages. It however does not hide the cost of communication, neither does it require global synchronization operations. Still, LogP assumes an absolute upper bound on communication delays, and thus models a *synchronous* distributed system.

Competitive analysis Other work, based on the method of competitive analysis proposed by Sleator and Tarjan [ST85], has focused on evaluating the competitiveness of distributed algorithms [AKP92, BFR92]. In this work, the cost of a distributed algorithm is compared to the cost of an optimal centralized algorithm with a global knowledge. This work has been refined in [AADW94, AW95, AW96] by considering an optimal *distributed* algorithm as the reference for the comparison. This work assumes an asynchronous shared-memory model and predicts the performance of an algorithm by counting the number of steps required by the algorithms to terminate. The idea of evaluating distributed algorithms against an optimal reference is appealing, but this approach is orthogonal to the definition of a metric. The metric used is designed for the shared-memory model, and totally ignores the problem of contention.

5.2 Distributed System Model

The two metrics that we define in this paper are based on an abstract system model which introduces two levels of resource contention: *CPU contention* and *network contention*. First, we define a basic version of the model that leaves some aspects unspecified, but is sufficient to define our throughput oriented metric (see Definition 5). Second, we define an extended version of the model by lifting the ambiguities left in the basic version. This extended model is used in Section 5.3 to define our latency oriented metric (see Definition 3).

5.2.1 Basic Model

The model is inspired from the models proposed in [Ser98, TBW95]. It is built around of two types of resources: CPU and network. These resources are involved in the transmission of messages between processes. There is only one network that is shared among processes, and is used to transmit a message from one process to another. Additionally, there is one CPU resource attached with each process in the system. These CPU resources represent the processing performed by the network controllers and the communication layers, during the emission or the reception of a message. In this model, the cost of running the distributed algorithm is neglected, and hence does not require any CPU resource.
The transmission of a message m from a sending process p_i to a destination process p_j occurs as follows (see Fig. 5.1):

1. m enters the *sending queue*[1] of the sending host, waiting for CPU$_i$ to be available.
2. m takes the resource CPU$_i$ for λ time units, where λ is a parameter of the system model ($\lambda \in \mathbb{R}_0^+$).
3. m enters the *network queue* of the sending host and waits until the network is available for transmission.
4. m takes the network resource for 1 time unit.

[1]All queues in the model use a FIFO policy (sending, receiving, and network queues).

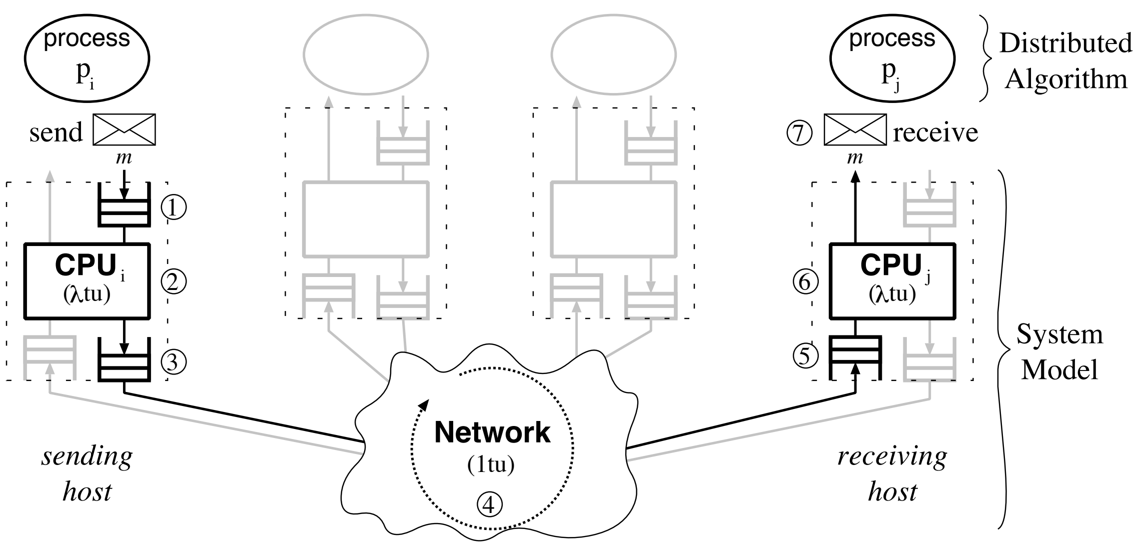


Figure 5.1: Decomposition of the end-to-end delay (tu=time unit)

5. m enters the *receive queue* of the destination host and waits until CPU_j is available.

6. m takes the resource CPU_j of the destination host for λ time units.

7. Message m is received by p_j in the algorithm.

5.2.2 Extended Model

The basic model is not completely specified. For instance, it leaves unspecified the way some resource conflicts are resolved. We now extend the definition of the model so that to specify these points. As a result, the execution of a (deterministic) distributed algorithm in the extended system model is *deterministic*.

Network Concurrent requests to the network may arise when messages at different hosts are simultaneously ready for transmission. The access to the network is modeled by a round-robin policy,[2] illustrated by Algorithm 5.1.

CPU CPU resources also appear as points of contention between a message in the sending queue and a message in the receiving queue. This issue is solved by giving priority on every host to outgoing messages over incoming ones.

Send to oneself It often happens that distributed algorithms require a process p to send a message m to itself. In such a case, the message is considered "virtual" and takes neither CPU nor network resources.

[2]Many thanks to Jean-Yves Le Boudec for this suggestion.

Algorithm 5.1 Network access policy (code executed by the network).

```
$i \leftarrow 1$
loop
  wait until one network queue is not empty
  while network queue of $\mathrm{CPU}_i$ is empty do
    increment $i$ (mod $n$)
  $m \leftarrow$ extract first message from network queue of $\mathrm{CPU}_i$
  wait 1 time unit
  insert $m$ into receiving queue of $\mathrm{CPU}_{\mathrm{dest}(m)}$
  increment $i$ (mod $n$)
```

Send to all Distributed algorithms often require to send a message m to all processes, using a "send to all" primitive. The way this is actually performed depends on the model (see below).

Definition 1 (point-to-point) *Model $\mathcal{M}_{\mathrm{pp}}(n,\lambda)$ is the extended model with parameters $n \in \mathbb{N}$ and $\lambda \in \mathbb{R}_0^+$, where $n > 1$ is the number of processes, and λ is the relative cost between CPU and network. The primitive "send to all" is defined as follows: If p is a process that sends a message m to all processes, then p sends the message m consecutively to all processes in the lexicographical order (p_1, p_2, ..., p_n).*

Nowadays, many networks are capable of broadcasting information in an efficient manner, for instance, by providing support for IP multicast [Dee89]. For this reason, we also define a model that integrates the notion of a broadcast network.

Definition 2 (broadcast) *Model $\mathcal{M}_{\mathrm{br}}(n,\lambda)$ is defined similarly to Definition 1, with the exception of the "send to all" primitive, which is defined as follows: If p is a process that sends a message m to all, then p sends a single copy of m, the network transmits a single copy of m, and each process (except p) receives a copy of m.*

5.2.3 Illustration

Let us now illustrate the model with an example. We consider a system with three processes $\{p_1, p_2, p_3\}$ which execute the following simple algorithm. Process p_1 starts the algorithm by sending a message m_1 to processes p_2 and p_3. Upon reception of m_1, p_2 sends a message m_2 to p_1 and p_3, and p_3 sends a message m_3 to p_1 and p_2.

Figure 5.2 shows the execution of this simple algorithm in model $\mathcal{M}_{\mathrm{br}}(3, 0.5)$. The upper part of the figure is a time-space diagram showing the exchange of messages between the three processes (message exchange, as seen by the distributed algorithm). The lower part is a more detailed diagram that shows the activity (send, receive, transmit) of each resource in the model. For instance, process p_1 sends a message m_1 to process p_2 and p_3 at time 0. The message takes the CPU resource of p_1 at time 0, takes the network resource at time 0.5, and takes the CPU resource of p_2 and p_3 at time 1.5. Finally, p_2 and p_3 simultaneously received m_1 at time 2.

Similarly, Figure 5.3 shows the execution of the same simple algorithm in model $\mathcal{M}_{\mathrm{pp}}(3, 0.5)$. The network is point-to-point, so wherever a message is sent to all, the model actually sends as

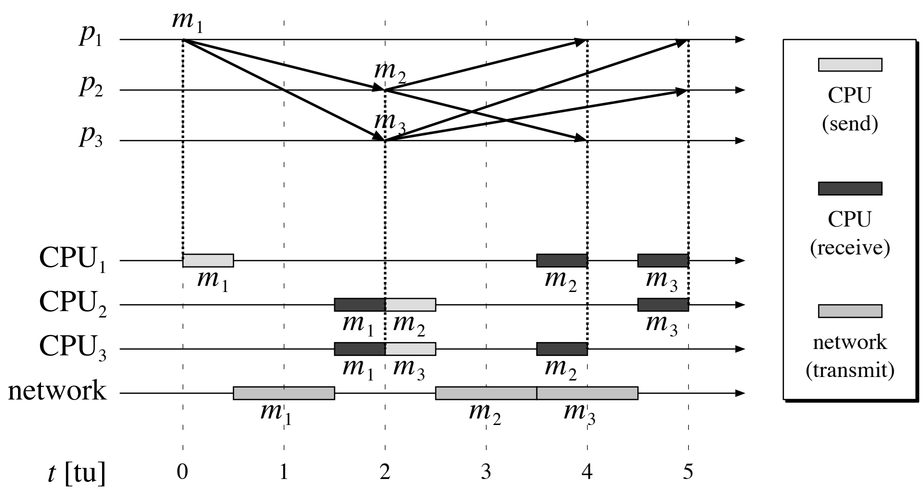


Figure 5.2: Simple algorithm in model $\mathcal{M}_{\mathrm{br}}(3, 0.5)$

many *copies* of that message. For instance, process p_3 sends a copy of message m_3 to process p_1 (denoted $m_{3,1}$) at time 3. Copy $m_{3,1}$ takes the CPU resource of p_3 at time 3, takes the network resource at time 4.5, and takes the CPU resource of p_1 at time 5.5. Finally, p_1 receives m_3 at time 6.

5.3 Contention-Aware Metrics

We now define a latency and a throughput metric based on the system models defined in Section 5.2. The latency metric is based on the extended model of Section 5.2.2. Conversely, the throughput metric is based on the simpler definition of the basic system model of Section 5.2.1.

5.3.1 Latency Metric

The definition of the latency metric uses the terms: "start" and "end" of a distributed algorithm. These terms are supposed to be defined by the problem $\mathcal{P}$ that an algorithm $\mathcal{A}$ solves. They are not defined as a part of the metric.

Definition 3 (latency metric) *Let $\mathcal{A}$ be a distributed algorithm. The definition of the latency metric* $\mathrm{Latency}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$ *is the number of time units that separate the start and the end of algorithm $\mathcal{A}$ in model $\mathcal{M}_{\mathrm{pp}}(n, \lambda)$.*

Definition 4 (latency metric (broadcast)) *Let $\mathcal{A}$ be a distributed algorithm. The definition of the latency metric* $\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$ *is the number of time units that separate the start and the end of algorithm $\mathcal{A}$ in model $\mathcal{M}_{\mathrm{br}}(n, \lambda)$ (same as Definition 3, but in model $\mathcal{M}_{\mathrm{br}}(n, \lambda)$)*

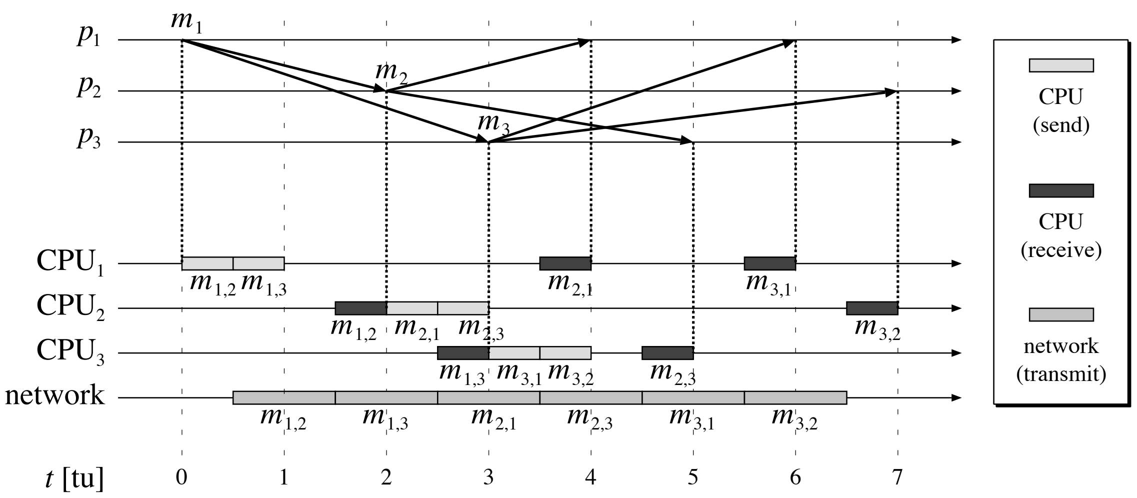


Figure 5.3: Simple algorithm in model $\mathcal{M}_{\text{pp}}(3, 0.5)$
($m_{i,j}$ denotes the copy of message m_i sent to process p_j)

5.3.2 Throughput Metric

The throughput metric of an algorithm $\mathcal{A}$ considers the utilization of system resources in one run of $\mathcal{A}$.[3] The most heavily used resource constitutes a bottleneck, which puts a limit on the *maximal throughput*, defined as an upper bound on the frequency at which the algorithm can be run.

Definition 5 (throughput metric) *Let $\mathcal{A}$ be a distributed algorithm. The throughput metric is defined as follows:*

$$\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n,\lambda) \overset{\mathrm{def}}{=} \frac{1}{\max_{r\in\mathcal{R}_n} T_r(n,\lambda)}$$

where $\mathcal{R}_n$ denotes the set of all resources (i.e., $\mathrm{CPU}_1, \ldots, \mathrm{CPU}_n$ and the network), and $T_r(n,\lambda)$ denotes the total duration for which resource $r \in \mathcal{R}_n$ is utilized in one run of algorithm $\mathcal{A}$ in model $\mathcal{M}_{\mathrm{pp}}(n,\lambda)$.

$\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n,\lambda)$ can be understood as an upper bound on the frequency at which algorithm $\mathcal{A}$ can be started. Let r_b be the resource with the highest utilization time: $T_{r_b} = \max_{r\in\mathcal{R}_n} T_r$. At the frequency given by $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n,\lambda)$, r_b is utilized at 100%, i.e., it becomes a bottleneck.

Definition 6 (throughput metric (broadcast)) *Let $\mathcal{A}$ be a distributed algorithm. The definition of the throughput metric $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$ is the same than Definition 5, but in model $\mathcal{M}_{\mathrm{br}}(n,\lambda)$.*

The meaning of the value computed by $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n,\lambda)$—or $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$—is normally comprised between 0 and 1. A throughput of 0 means that not a single execution of algorithm $\mathcal{A}$ terminates. In other words, this means that one execution of $\mathcal{A}$ monopolizes one of the resources for an infinite time, thus preventing a second execution to start.

[3]Our thanks to Jean-Yves Le Boudec for this suggestion.

Conversely, a throughput value of 1 means that each execution of algorithm $\mathcal{A}$ does not use any resource for more than 1 time unit. This can also be understood as a factor of comparison with the following simple reference algorithm $\mathcal{A}_0$: process p_i sends one message m to a process p_j; the algorithm terminates once p_j has received m.

Relation with message complexity The throughput metric can be seen as a generalization of message complexity. While our metric considers different types of resources, message complexity only considers the network. It is easy to see that $T_{network}$, the utilization time of the network in a single run, gives the number of messages exchanged in the algorithm.

5.4 Summary

In the field of parallelism, researchers have seen early the limitations of simple complexity metrics to anticipate the performance of parallel algorithms. Driven by the need to better understand the impact of their choices in algorithm design, they have developed several system models in which to analyze the performance of algorithms (e.g., PRAM, BSP, LogP).

In the context of distributed algorithms, most researchers continue to rely solely on the result of complexity metrics, and have not yet seen the importance of developing more accurate performance metrics. The performance models developed for parallel algorithms are poorly suited to evaluate the performance of distributed algorithms. For this reason, it is necessary to develop metrics that are better adapted to distributed algorithms.

The problem of resource contention is commonly recognized as having a major impact on the performance of distributed algorithms. This chapter presents two metrics to predict the latency and the throughput of distributed algorithms. Unlike other more conventional metrics, the two metrics that are introduced here take account of both network and CPU contention. This allows for more precise predictions and a finer grained analysis of algorithms than what time complexity and message complexity permit.

These metrics are used in Chapter 6 to analyze total order broadcast algorithms. These performance analyses also provide an extensive example of the use of the contention-aware metrics.

Chapter 6

Evaluation of Total Order Broadcast Algorithms

Everything happens to everybody sooner or later if there is time enough.

— **George Bernard Shaw** (1856–1950)

The classification of total order broadcast algorithms presented in Chapter 4 provides a *qualitative* comparison of the different total order broadcast algorithms. In contrast, we now use the contention-aware metrics defined in Chapter 5 to provide a *quantitative* comparison of total order broadcast algorithms.

The effort required for computing the metrics for every single algorithm published so far is of course totally disproportionate, and would ask for a more precise definition of the algorithms than what is available. Besides, such an analysis is likely to bring a large amount of raw information, but very little insight. To overcome this problem, we use the classification of Chapter 4 and define algorithms to represent each *class* of algorithms. We then compare these representative algorithms using our metrics and thus derive general observations. The results obtained are more precise than what can be obtained by relying solely on time and message complexity, and more general than measures obtained through simulation.

The rest of the chapter is structured as follows. Section 6.1 defines the representative algorithms. Section 6.3 evaluates and compares the latency of the representative algorithms. Section 6.4 compares the throughput of the algorithms. Finally, Section 6.5 consolidates the observations made in the two previous sections, and concludes the chapter.

6.1 Representative Algorithms for Total Order Broadcast

This section defines representative algorithms for each class. The representative algorithms are mostly derived from the algorithms presented in Section 4.3. Some of the algorithms are also inspired from actual algorithms that belong to the corresponding class.

Unless stated otherwise, all representative algorithms are broadcast algorithms (i.e., messages are targeted at every process in the system). They are based on the following system model:

- Processes can only fail by crashing.
- Communication channels are reliable and FIFO.[1]
- The management of failure is handled by an adequate group membership service.

The group membership is left unspecified because we do not analyze the case when failures (or suspicions) occur. This is left aside because such an analysis would compare different group membership implementations rather than different algorithms.[2]

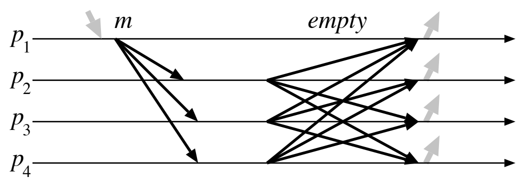
Wherever possible, we consider two representatives for each class: a uniform algorithm and a non-uniform one. A *uniform algorithm* means that the algorithm satisfies Uniform Integrity, Uniform Agreement, and Uniform Total Order (see Sect. 4.1, p.50). A *non-uniform algorithm* relinquishes one or more of the uniform properties for the sake of performance (it usually improves latency rather than throughput).

In order to improve readability, we only give an informal description of the algorithms, and illustrate their execution on a time-space diagram. Each of these diagrams illustrates an execution of the corresponding algorithm, in which one process broadcasts a message m in isolation. Note that the communication pattern is sufficient to compute the metrics. The pseudo-code of each algorithm is also given for reference in Appendix A.

6.1.1 Communication History

There is only one representative algorithm for the class of communication history algorithms. This algorithm is uniform, as is the case with most (non-Byzantine) algorithms that belong to this class (see Sect. 4.5.1).

The algorithm we consider here is derived from Algorithm 4.1 (p.60). The major extension lies with the use of *empty* messages to guarantee the liveness of the algorithm. When a process has sent no message for a certain time, it sends an "*empty*" message. This behavior is common among algorithms of this class (e.g., [EMS95]).



(a) Uniform delivery

Figure 6.1: Representative for communication history algorithms

[1]The communication history algorithm is the only one that actually requires FIFO communication.
[2]See Sect. 7.2 for a more detailed discussion on this issue.

Figure 6.1 illustrates a run of the algorithm in which one process (p_1) broadcasts a single message. The algorithm is based on Lamport's logical clocks [Lam78].

Informally, the algorithm runs as follows. Every message is timestamped using Lamport's logical clocks. Every process p manages a vector of logical clocks LC_p, where $LC_p[p]$ represents the logical clock of process p, and $LC_p[q]$ ($q \neq p$) is p's most recent estimation of the value q's own logical clock. When process p broadcasts a message m, its timestamp $ts(m)$ is equal to the timestamp of the $send(m)$ event. When a process q receives m from p, it buffers m, and updates $LC_q[p]$ and $LC_q[q]$. For process q, message m becomes a *deliverable message* when its satisfies the following condition:

$$\forall k, ts(m) < LC_q[k]$$

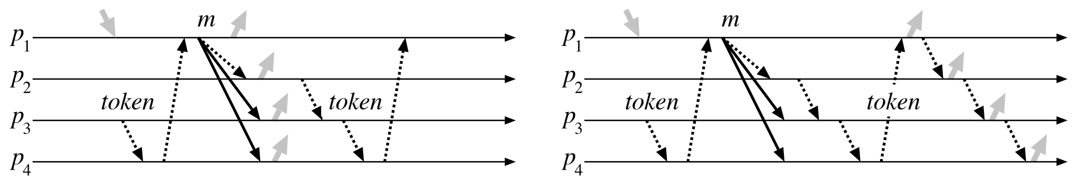
Because of FIFO channels, this means that no process can send an other message with a timestamp that is smaller or equal to $ts(m)$. Deliverable messages are then delivered according to the total order defined by the relation "$\Longrightarrow$" described in Section 4.3.1.

As such, the algorithm would not be live. Indeed, the algorithm relies on the assumption that all processes communicate often with each other. If this is not the case (e.g., only one process broadcasts messages), then the algorithm is not live. To overcome this problem, we consider the following extension. If a process p has not sent any message after a delay Δ_{live}, then p sends an *empty* message to all other processes. For the purpose of computing the metrics, we avoid this delay by introducing the following bias: $\Delta_{\text{live}} = 0$. This means that there is no such delay when computing the latency. Also, due to the scenario used when computing the throughput, the algorithm does not generate empty messages nevertheless.

6.1.2 Privilege-Based

privilege-based algorithms have two representative algorithms: a uniform algorithm and a non-uniform one. In fact, both algorithms follow the same communication pattern. The only difference is that the non-uniform algorithm can deliver messages earlier.

The algorithms are derived from Algorithm 4.2 (p.60), and are also largely influenced by TPM [RM89] and Totem [AMMS+95].



(a) Non-uniform delivery

(b) Uniform delivery

Figure 6.2: Representative for privilege-based algorithms

Figure 6.2 illustrates a run of each algorithm in which one process (p_1) broadcasts a single message. The messages that carry the token are represented by a dashed arrow, whereas plain messages are represented by a solid arrow.

Informally, the non-uniform algorithm (see Fig.6.2(a)) works as follows. When a process p wants to broadcast a message m, it simply stores m into a send queue until it receives the token. The token carries a sequence number and constantly circulates among the processes. When p receives the token, it extracts m from its send queue, uses the sequence number carried by the token, and broadcasts m with the sequence number. Then, p increments the sequence number and propagates the token to the next process. Taking inspiration from Totem, p broadcasts the token along with m. In other words, it piggybacks the token on m. When a process q receives m, it delivers m according to its sequence number.

In the uniform algorithm (Fig.6.2(b)), a process can deliver a message m only once it knows that all other processes have received m (i.e., when m is stable). This is done by extending the non-uniform algorithm in the following way. In addition to the sequence number, the token also carries acknowledgements. A process q can deliver m when (1) q has received m and its sequence number, (2) q has deliver every message with a smaller sequence number, and (3) m has been acknowledged by all processes. This last condition requires that the token performs at least a full round trip before m can be delivered.

privilege-based algorithms are usually parametrized as follows: when a process receives the token, it is allowed to broadcast as most k messages before it must relinquish the token. Determining the ideal value for k is not easy. On the one hand, a large value means that other processes must wait longer to get the token.[3] On the other hand, a small value reduces the opportunities for piggybacking. In this analysis, we have chosen to set the parameter k to 1.

Uniformity *vs.* non-uniformity Figure 6.3 depicts a scenario that illustrates the non-uniformity of the algorithm in Figure 6.2(a). The scenario initially involves a group with five processes. At some point, process p_1 wants to broadcast a message m. It thus stores m into its send queue and waits for the token. Once p has received the token, it assigns a sequence number to m, and sends m along with the sequence number to the other processes. However, p_1 crashes while sending m with its sequence number. As a result, only process p_2 receives m and the sequence number. According to the algorithm, p_2 delivers m. But, p_2 also crashes shortly after, before sending the token to p_3. Some time later, both crashes are detected and a new group is formed, that excludes both p_1 and p_2. After the crash of p_1 and p_2, there is no copy of m that can be retransmitted. Consequently, none of the correct processes (i.e., p_3, p_4, p_5) is able to deliver m despite the fact that p_2 has delivered it, thus violating the property of Uniform Agreement. Similar arguments can be used for the non-uniform algorithms of the two sequencer classes (moving and fixed sequencer; see below).

6.1.3 Moving Sequencer

Moving sequencer algorithms are represented by both a uniform and a non-uniform algorithm. The non-uniform algorithm can deliver messages earlier but, unlike privilege-based algorithms,

[3]It may even lead to starvation if the parameter k is infinite.

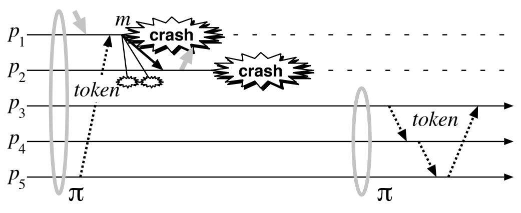
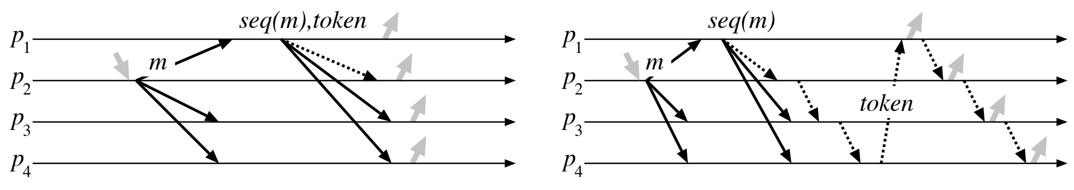


Figure 6.3: Non-uniform agreement with a privilege-based algorithm

the delivery of messages is not the only difference between the uniform and the non-uniform algorithms.[4]

Although the chosen algorithms are based on the principle illustrated by Algorithm 4.3 (p.62), the resulting algorithms are not too similar. The representative algorithms are inspired from both Chang and Maxemchuck's algorithm [CM84] and Pinwheel [CMA97, CM95].



(a) Non-uniform delivery

(b) Uniform delivery

Figure 6.4: Representative for moving sequencer algorithms

Figure 6.4 illustrates a run of each algorithm in which process p_3 broadcasts a message m. Similar to the privilege-based algorithms, the messages that carry the token are represented by a dashed arrow. The token also carries a sequence number for numbering messages.

In short, the non-uniform algorithm (Fig.6.4(a)) works as follows. When a process p wants to broadcast a message m, it sends m to all other processes. Upon receiving m, processes store it into a receive queue. When the current token holder q has a message in its receive queue, it uses the sequence number to number the first message in the queue and broadcasts that sequence number together with the token. A processes can then deliver m when it has (1) received m, (2) received m's sequence number, and (3) delivered every message with a smaller sequence number.

The uniform algorithm (Fig.6.4(b)) works pretty much the same. However, a process must wait until a message m is stable before it can deliver it. Similar to the uniform representative for privilege-based algorithms, the detection of stability is also done with the token.

In the moving sequencer algorithm described here, the token holder handles a single message before it passes on the token. An alternative would allow the token holder to give a sequence

[4]Indeed, in the moving sequencer algorithms, the token does not need to rotate all the time, and thus the token stops when it does not need to rotate. The uniform and the non-uniform algorithms differ with the moment at which the token can stop rotating. In the non-uniform algorithm, the token can stop earlier, thus generating less contention.

number to every undelivered message that waits in its receive queue. However, the algorithm would then run in a non-deterministic manner, thus preventing the computing of the metrics.

6.1.4 Fixed Sequencer

Fixed sequencer algorithms described in the literature are rarely uniform. The main reason is probably that uniformity implies a higher cost for fixed sequencer algorithms than for those of other classes. Fixed sequencer algorithms have nevertheless both a non-uniform and a uniform representative algorithm.

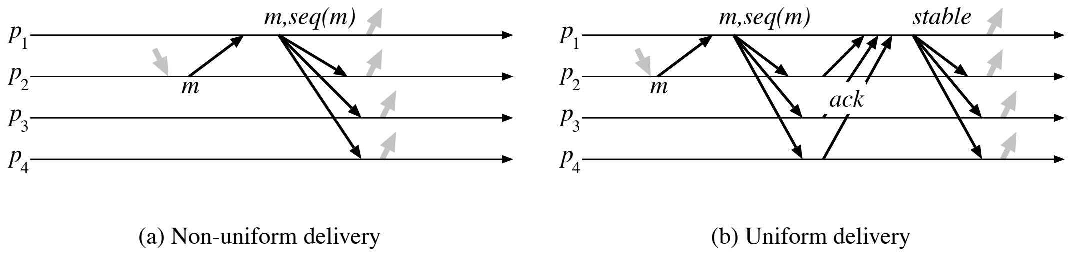
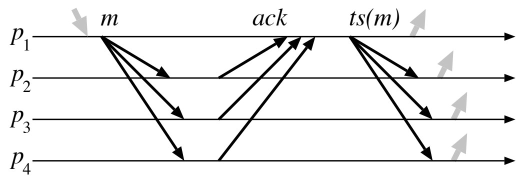


Figure 6.5: Representative for fixed sequencer algorithms

The representative algorithms are derived from Algorithm 4.3.4 (p.62) and follow the approach illustrated in Figure 4.9(a) (p.76). The representative algorithms run as follows: when a process p wants to broadcast a message m, it sends m to the sequencer. The sequencer assigns a sequence number to m, and sends both m and the sequence number to the other processes. In the non-uniform algorithm, processes deliver m as soon as they receive it with its sequence number (Fig.6.5(a)). In the uniform algorithm (Fig.6.5(b)), the processes can deliver m only after it is stable. The detection of stability is coordinated by the sequencer, as shown on Figure 6.5(b).

6.1.5 Destinations Agreement

The class of destinations agreement algorithms is only represented by a non-uniform algorithm, inspired from Skeen's algorithm (see below).



(a) Non-uniform delivery

Figure 6.6: Representative for destinations agreement algorithms

Similar to Algorithm 4.5 (p.64), the algorithm illustrated in Figure 6.6 is largely inspired from Skeen's algorithm, as described by Guerraoui and Schiper [GS97c]. The main difference with Algorithm 4.5 is that this algorithm follows a *centralized* communication pattern rather than a decentralized one (see Fig.4.10, p. 80).

When a process p wants to broadcast a message m, it sends m to all processes and acts as a coordinator for the delivery of m. Processes manage a counter as some form of logical clock. Upon receiving m, a process q sends an acknowledgement back to p, together with a logical temporary timestamp $ts_q(m)$. Once p has collected an acknowledgement from all other processes, it takes the maximum of all received timestamps as the final timestamp $TS(m)$ for message m. Then, it sends $TS(m)$ to all processes. A process q can deliver m once (1) it has received m and its final timestamp $TS(m)$, (2) it has delivered every message with a smaller final timestamp, and (3) there is no message m' waiting for its final timestamp $TS(m')$, with a temporary timestamp $ts_q(m')$ smaller than $TS(m)$.

Note on uniform destinations agreement algorithms According to Section 4.5.5, many destinations agreement algorithms are uniform. However, the definition of a representative algorithm for this class poses many problems.

Indeed, the class of destinations agreement algorithms is by far the most diverse. Although this diversity is not visible with non-uniform algorithms (Skeen's algorithm is the only representative), this is different with uniform algorithms. Uniform destinations agreement algorithms often differ with the way they solve subproblems. For instance, Chandra and Toueg's algorithm [CT96] (see Sect. 4.5.5) is based on a Consensus algorithm. But, there exist a very large number of algorithms to solve Consensus.

As a result, a study of uniform destinations agreement algorithms is only possible after all tradeoffs are identified. The large number of possible combinations stands clearly as an obstacle to such an analysis. To be manageable, this requires to decompose the algorithms into several building blocks, in order to first analyze them in isolation.

This decomposition of destinations agreement algorithms is an interesting problem but goes far beyond the scope of this dissertation. For this reason, we do not define a representative algorithm for uniform destinations agreement algorithms, and keep this study for future work.

6.2 Conventional Complexity Measures

In order to compare the representative algorithms, we present their respective time and message complexity in Table 6.1.

To determine the time complexity of the algorithms, we use the definition of *latency degree* as defined by Schiper [Sch97]: roughly speaking, an algorithm with latency degree l requires at least l communication steps to terminate.

The message complexity presented in Table 6.1 counts the number of *protocol messages* generated when a single process decides to TO-broadcast a single *application message*.

Note that we give two values for message complexity. The reason is that message complexity

Table 6.1: Complexity measures of total order broadcast algorithms

Algorithms		**Time complexity**	**Message complexity**	
			point-point	**broadcast**
Uniform	**Communication history**	2	$n^2 - n$	n
	privilege-based[a]	$\frac{5n}{2}$	$\frac{7n}{2} - 3$	$\frac{5n}{2} - 2$
	Moving sequencer	$2n$	$4n - 4$	$2n$
	Fixed sequencer	2	$3n - 2$	$n + 2$
Non-Uniform	**privilege-based**[a]	$\frac{n}{2} + 1$	$\frac{3n}{2} - 1$	$\frac{n}{2} + 1$
	Moving sequencer	2	$2n - 2$	2
	Fixed sequencer	2	n	2
	Destinations agreement	3	$3n - 3$	$n + 1$

[a]The privilege-based algorithms (uniform and non-uniform) generate a load even when there is no message to order. This is due to the necessity of constantly circulating the token among the processes.

is a measure that depends on the communication model. The value on the left is the message complexity in a point-to-point network, where a "send to all" yields $n - 1$ messages. Conversely, the value on the right is the message complexity in a broadcast network, where a "send to all" generates just a single message.

In Table 6.1, the time complexity of the uniform algorithms give that the communication history and the fixed sequencer algorithms have the best latency (latency degree 2). At the same time, the other two uniform algorithms (privilege-based and moving sequencer) have a bad latency (latency degree $\approx 2n$).

Judging from the message complexity in point-to-point networks, the communication history has the worst throughput of all uniform algorithms (message complexity is $O(n^2)$). The other uniform algorithms have a throughput similar to each other (message complexity is $O(n)$). In a broadcast network, the message complexity of the communication history algorithm is linear, and the best of all uniform algorithms.

In Table 6.1, the time complexity of non-uniform algorithms yields that the privilege-based algorithm has by far the worst latency. Indeed, this is the only non-uniform algorithm with a latency degree that increases (linearly) with n.

In point-to-point networks, the message complexity of non-uniform algorithms yields that the fixed sequencer algorithm has the best throughput, while the destinations agreement algorithm has the worst one. According to the message complexity in broadcast networks, the two sequencer algorithms (fixed and moving) have the best throughput.

6.3 Latency Analysis with Contention-Aware Metrics

We now analyze the latency of the representative algorithms using the metrics defined in Chapter 5. First, we compare the four uniform algorithms: *communication history*, *uniform privilege-based*, *uniform moving sequencer*, and *uniform fixed sequencer*. We then discuss the four non-uniform algorithms: *non-uniform privilege-based*, *non-uniform moving sequencer*, *non-uniform fixed sequencer*, and *destinations agreement*.

For the sake of readability, we only give a graphical representation of the results obtained with the metric. The exact formulas are given in Appendix A.

6.3.1 Latency of Uniform Algorithms

We analyze the latency of the uniform algorithms: *communication history*, *uniform privilege-based*, *uniform moving sequencer*, and *uniform fixed sequencer*. We first consider these algorithms in a point-to-point network, and illustrate the results of $\mathrm{Latency}_{\mathrm{pp}}(\mathcal{A})(n,\lambda))$ in Figure 6.7. Then, we analyze the same algorithms in a broadcast network using $\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$, and show the results in Figure 6.8.

Latency of Uniform Algorithms in Point-to-Point Networks

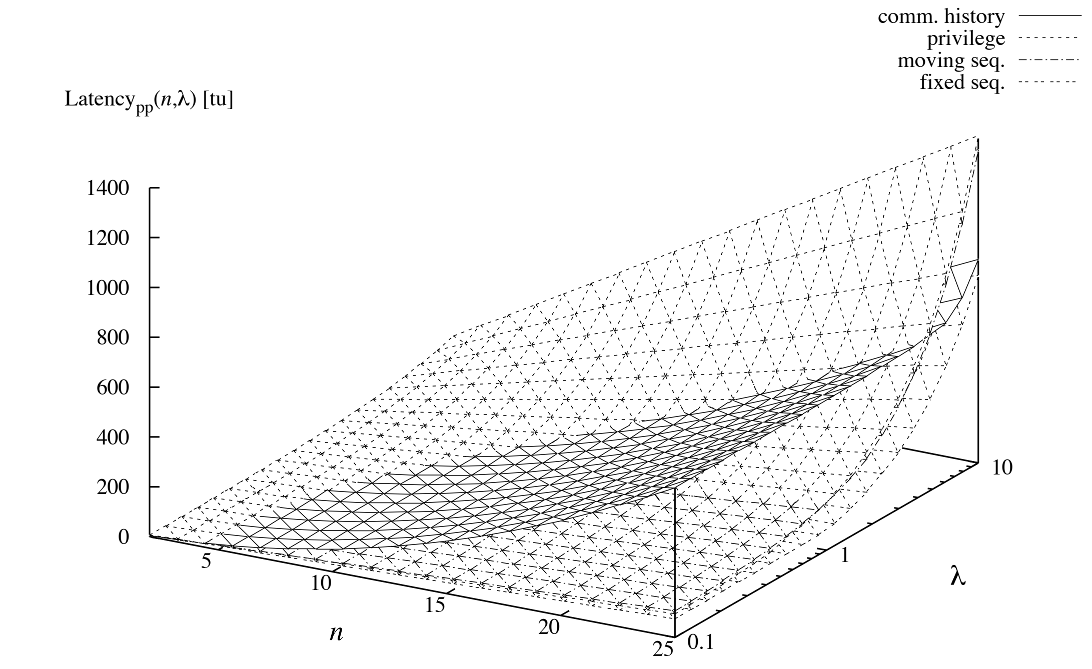
Figure 6.7(a) represents the latency of the four uniform algorithms according to the two parameters of the model: n and λ. Each surface represents one algorithm. Figure 6.7(b) is a cross-section of Figure 6.7(a) along the n-axis; i.e., it shows the evolution of the latency of each algorithm when n varies. In the figure, the parameter λ is set to 1, thus representing a system in which CPU and network are equivalently important. Figure 6.7(c) is also a cross-section of Figure 6.7(a), but along the λ-axis, for $n = 10$.

According to the conventional metrics (Table 6.1), the fixed sequencer and the communication history algorithms have the best latency (latency degree of 2). Then, the moving sequencer and privilege-based algorithms both have a worse latency. This conclusion seems simplistic when compared with the results plotted on Figure 6.7.

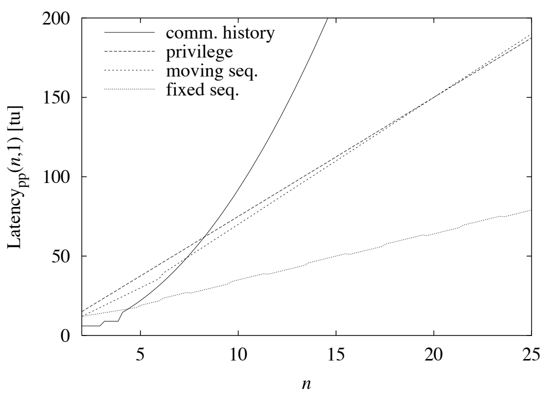
As a first observation, Figure 6.7(b) shows that the quadratic number of messages generated by the communication history algorithm causes a lot of network contention, and has a strong influence on the latency. For instance, because of this contention, the communication history algorithm has the worst latency in a system with more than 10 processes. In contrast, this algorithm has the best latency in a system with less than five processes. This is easily explained by the fact that, despite the quadratic number of messages, the algorithm generates only little network contention with such a small number of processes.

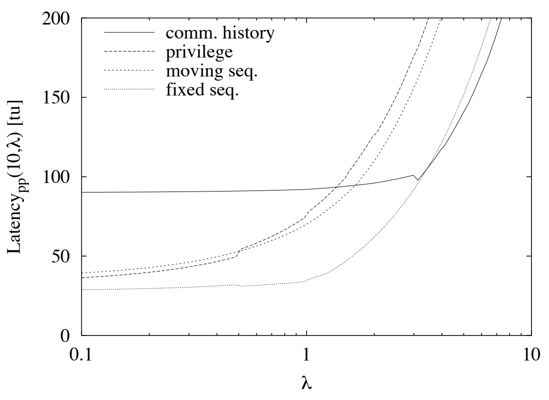
As a second observation, Figure 6.7(c) shows something unexpected: the communication history algorithm has the best latency of all four algorithms, for large values of λ (larger than 4 for $n = 10$). A plausible explanation is that the communication history algorithm has a decentralized communication pattern. This increases the parallelism between processes and thus reduces potential waiting delays. Also, a large value of λ reduces the importance of the network on the overall performance. The network contention is thus the weak point of the communication history algorithm.

The third observation is less obvious, and concerns the privilege-based and the moving sequencer algorithms. The two algorithms have a similar latency. In general, the moving sequencer algorithm seems to perform slightly better than the privilege-based algorithm; the opposite of what



(a) Uniform algorithms ($\mathrm{Latency_{pp}}(\mathcal{A})(n, \lambda)$)





(b) $\mathrm{Latency_{pp}}(\mathcal{A})(n, \lambda = 1)$

(c) $\mathrm{Latency_{pp}}(\mathcal{A})(n = 10, \lambda)$

Figure 6.7: Graphical representation of $\mathrm{Latency_{pp}}(\mathcal{A})(n, \lambda)$ for uniform algorithms

time complexity suggests. Nevertheless, Figure 6.7(a) and Figure 6.7(c) show that, for small values of λ, this is reversed and the privilege-based algorithm has a slightly better latency than the moving sequencer algorithm. Figure 6.7(b) shows that the privilege-based algorithm has a better latency also when the number of processes becomes large ($n > 20$ when $\lambda = 1$).

Latency of Uniform Algorithms in Broadcast Networks

As for Figure 6.7, Figure 6.8 is also made of three parts. Figure 6.8(a) depicts the evolution of the latency for varying values of n and λ. Figure 6.8(b) and Figure 6.8(c) plot the latency when n varies and when λ varies, respectively.

The predictions obtained with $\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$ are globally closer to those based on time complexity. The reason is that the algorithms generate much less traffic than in a point-to-point network, and thus cause only little network contention.

As a general observation, the respective performance of the four algorithms does not seem to depend on either parameters (n or λ). The communication history algorithm has the best latency, while the privilege-based algorithm has the worst one.

It is interesting to note that, in a broadcast network, the communication history algorithm has a better latency than even the fixed sequencer algorithm. Fixed sequencer are normally considered to have the best latency, but this is mostly because one usually consider the non-uniform algorithm. Here, we see that the cost of uniformity is penalizing for fixed sequencer algorithms. Figure 6.8(b) shows that the communication history and the fixed sequencer algorithms have a better scalability than the two other algorithms.

A comparison between Figure 6.8 and Figure 6.7 clearly confirms that the communication history algorithm is the algorithm that most benefits from a broadcast network. This is understandable: the algorithm generates a linear number of messages in a broadcast network, instead of a quadratic number in a point-to-point network.

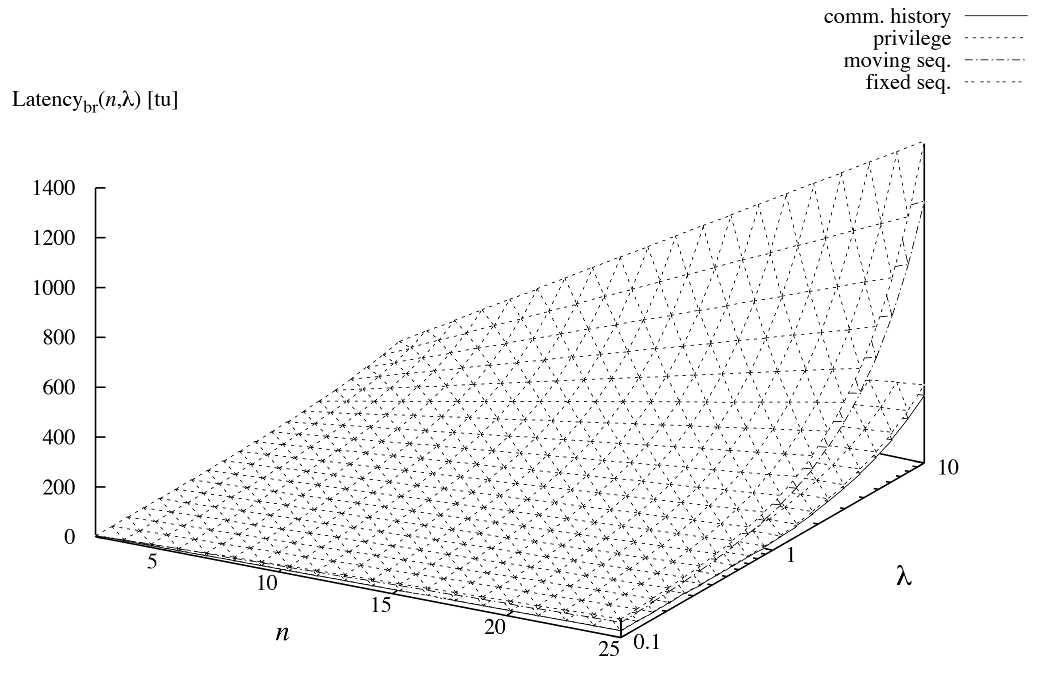
6.3.2 Latency of Non-Uniform Algorithms

We now analyze the latency of the non-uniform algorithms: *non-uniform privilege-based*, *non-uniform moving sequencer*, *non-uniform fixed sequencer*, and *destinations agreement*. As for the uniform algorithms, we first consider them in a point-to-point network, and then in a broadcast network. Figure 6.9 illustrates $\mathrm{Latency}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$, and Figure 6.10 illustrates $\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$.

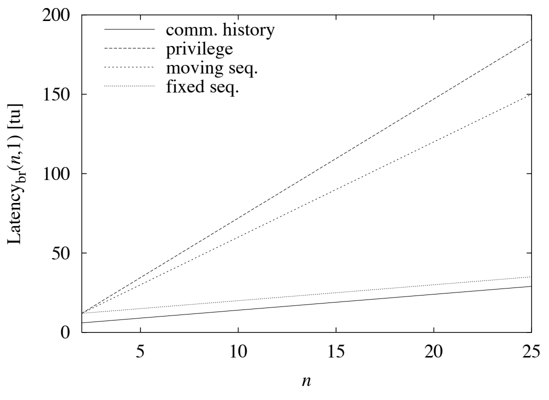
Latency of Non-Uniform Algorithms in Point-to-Point Networks

Figure 6.9(a) depicts the evolution of the latency for the four non-uniform algorithms in a point-to-point network. Figure 6.9(b) and Figure 6.9(c) plot the latency when n varies and when λ varies, respectively.

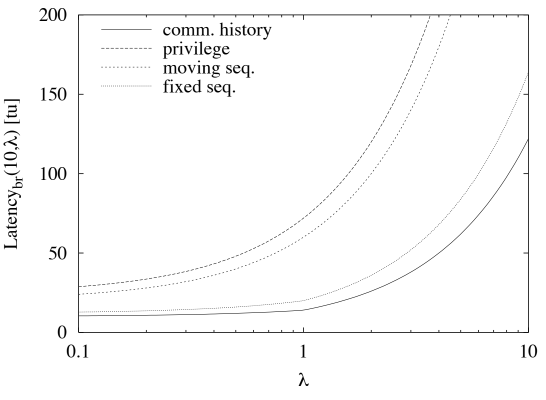
It is obvious from Figure 6.9 that the fixed sequencer algorithm has the best latency, while the destination agreement has the worst one. The privilege-based algorithm is a little better than



(a) Uniform algorithms ($\text{Latency}_{\text{br}}(\mathcal{A})(n, \lambda)$)

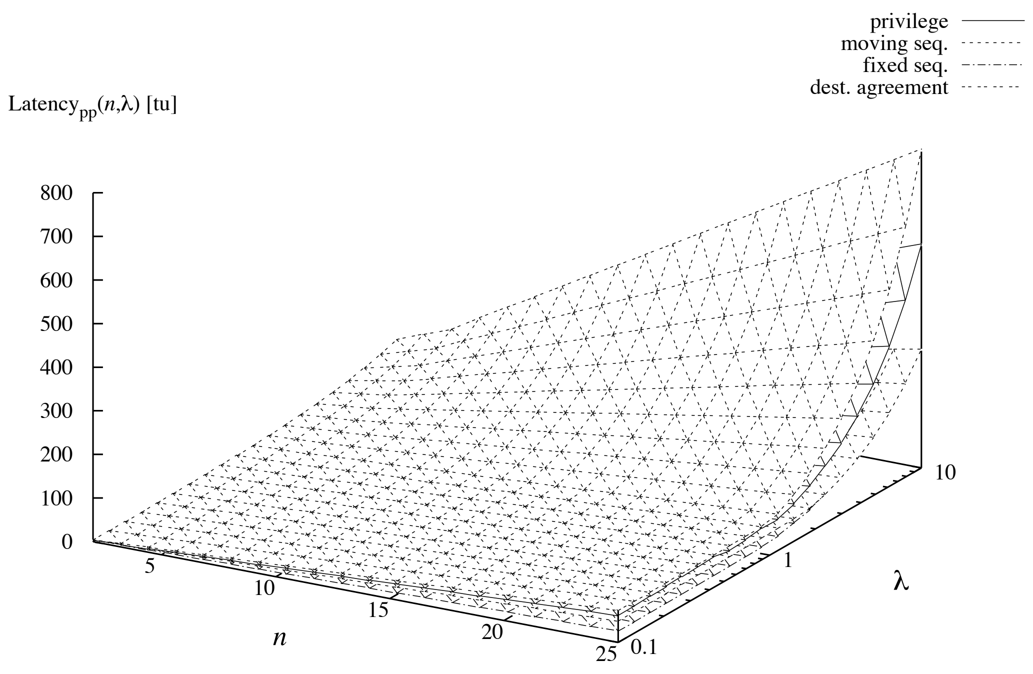


(b) $\text{Latency}_{\text{br}}(\mathcal{A})(n, \lambda = 1)$

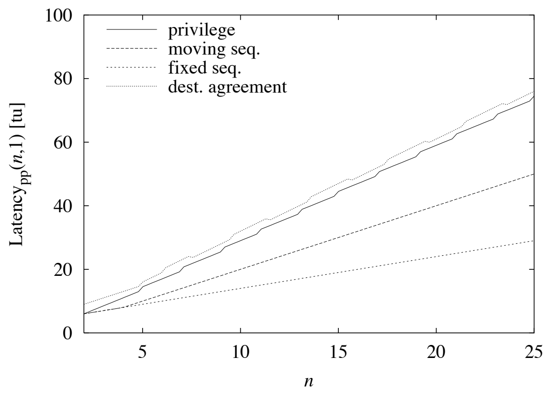


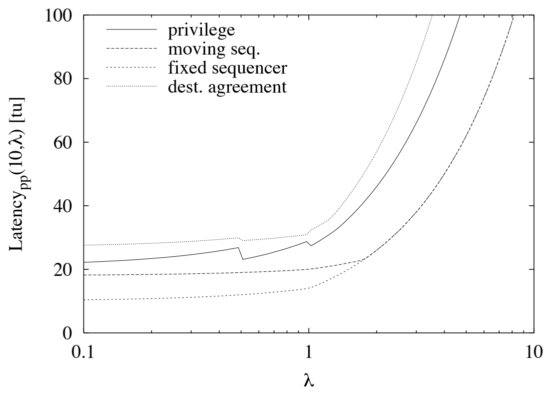
(c) $\text{Latency}_{\text{br}}(\mathcal{A})(n = 10, \lambda)$

Figure 6.8: Graphical representation of $\text{Latency}_{\text{br}}(\mathcal{A})(n, \lambda)$ for uniform algorithms



(a) Non-uniform algorithms ($\text{Latency}_{\text{pp}}(\mathcal{A})(n,\lambda)$)





(b) $\text{Latency}_{\text{pp}}(\mathcal{A})(n,\lambda=1)$

(c) $\text{Latency}_{\text{pp}}(\mathcal{A})(n=10,\lambda)$

Figure 6.9: Graphical representation of $\text{Latency}_{\text{pp}}(\mathcal{A})(n,\lambda)$ for non-uniform algorithms

the destinations agreement algorithm, whereas their respective time complexity suggest that the former is by far the worst. Also, the moving sequencer algorithm is just a little less efficient than the fixed sequencer one.

The latency of the moving sequencer algorithm is only a little less good than the latency of the fixed sequencer algorithm. This is especially true when λ is big, where both algorithms have the same latency (see Fig.6.9(c)). This is explained by the smaller importance of network contention for large values of λ.

Latency of Non-Uniform Algorithms in Broadcast Networks

Figure 6.10(a) shows the latency for the non-uniform algorithms in a broadcast network. Figure 6.10(b), resp. Figure 6.10(c), plots this evolution when n varies, resp. λ varies. In a broadcast network, the latency of the moving sequencer and the fixed sequencer algorithms are equal. For this reason, both algorithms are plotted together.

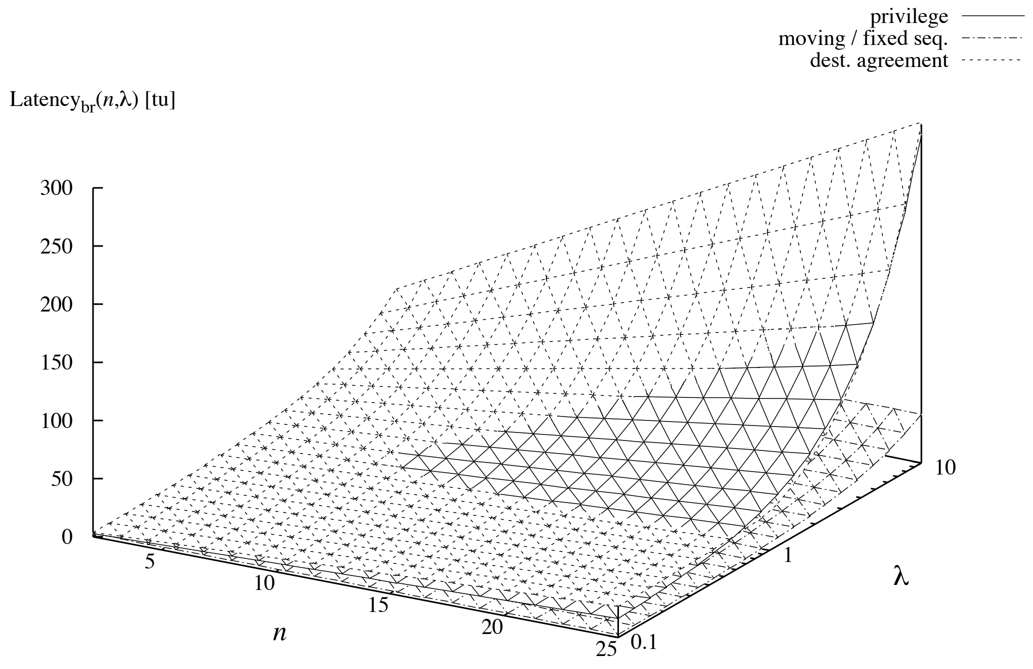
It is clear from Figure 6.10 that the sequencer algorithms (fixed and moving) have the best latency. The extra overhead that the moving sequencer algorithm has to pay for the first broadcast in a point-to-point network, disappears in a broadcast network. As a result, both sequencer algorithms have the same latency.

Figure 6.10 shows an interesting comparison between the privilege-based and the destinations agreement algorithms. First, Figure 6.10(b) clearly shows that the destinations agreement algorithm has the best latency when n is large. But, the relative performance of the algorithms when λ varies is even more interesting, as shown in Figure 6.10(a) and Figure 6.10(c). The privilege-based algorithm performs better than the destinations agreement for small values of λ, but also for large values of λ. The reason is that the destinations agreement algorithm benefits from some parallelism between the use of the network and CPU resources.

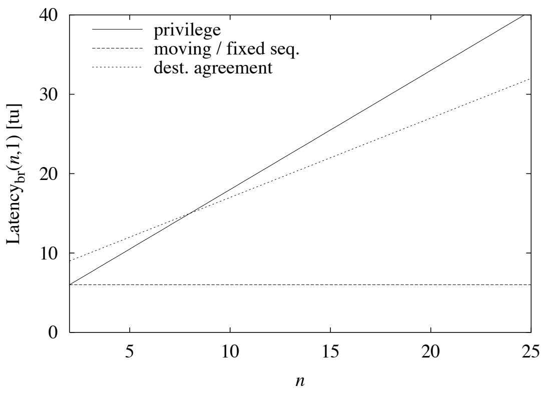
6.4 Throughput Analysis with Contention-Aware Metrics

We analyze the throughput of the representative algorithms. In a throughput analysis, one run of the algorithm is not considered in isolation. Indeed, many algorithms behave differently whether they are under high load or not (e.g., the representative for communication history does not need to generate empty messages under high load). For this reason, both the throughput metric and the corresponding message complexity are computed by considering a run of the algorithm *under high load*. We also assume that every process broadcasts messages, and that the emission is evenly distributed among them.

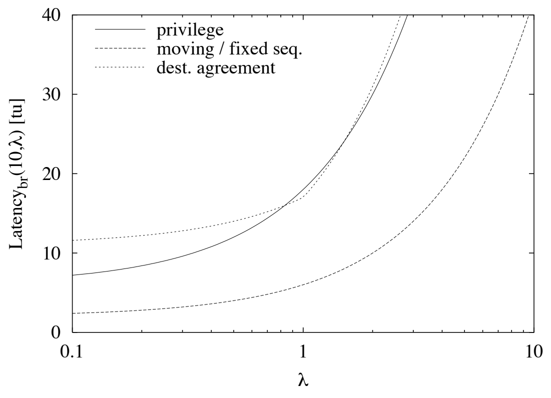
Table 6.2 gives the message complexity of the algorithms. The message complexity presented in this table is different from the message complexity of Section 6.2. Indeed, in Table 6.2, the message complexity counts the average number of *protocol messages* generated on behalf of a single *application* message, in a system under *high load*. When evaluating the throughput of algorithms, the results obtained are thus more realistic than those given in Section 6.2.



(a) Non-uniform algorithms ($\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$)



(b) $\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n, \lambda = 1)$



(c) $\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n = 10, \lambda)$

Figure 6.10: Graphical representation of $\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$ for non-uniform algorithms

Table 6.2: Messages complexity of total order broadcast algorithms *under high load*

Algorithms		**Message complexity**	
		point-point	**broadcast**
Uniform	**Communication history**	$n-1$	1
	privilege-based	$n-1$	1
	Moving sequencer	$2n-2$	2
	Fixed sequencer	$3n-2-\frac{1}{n}$	$n+2-\frac{1}{n}$
Non-Uniform	**privilege-based**	$n-1$	1
	Moving sequencer	$2n-2$	2
	Fixed sequencer	$n-\frac{1}{n}$	$2-\frac{1}{n}$
	Destinations agreement	$3n-3$	$n+1$

6.4.1 Throughput of Uniform Algorithms

We analyze the throughput of the uniform algorithms: *communication history*, *uniform privilege-based*, *uniform moving sequencer*, and *uniform fixed sequencer*. First, we compute the throughput in a point-to-point network ($\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n,\lambda)$), and show the results in Figure 6.11. We then illustrate the results obtained with $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$ in Figure 6.12.

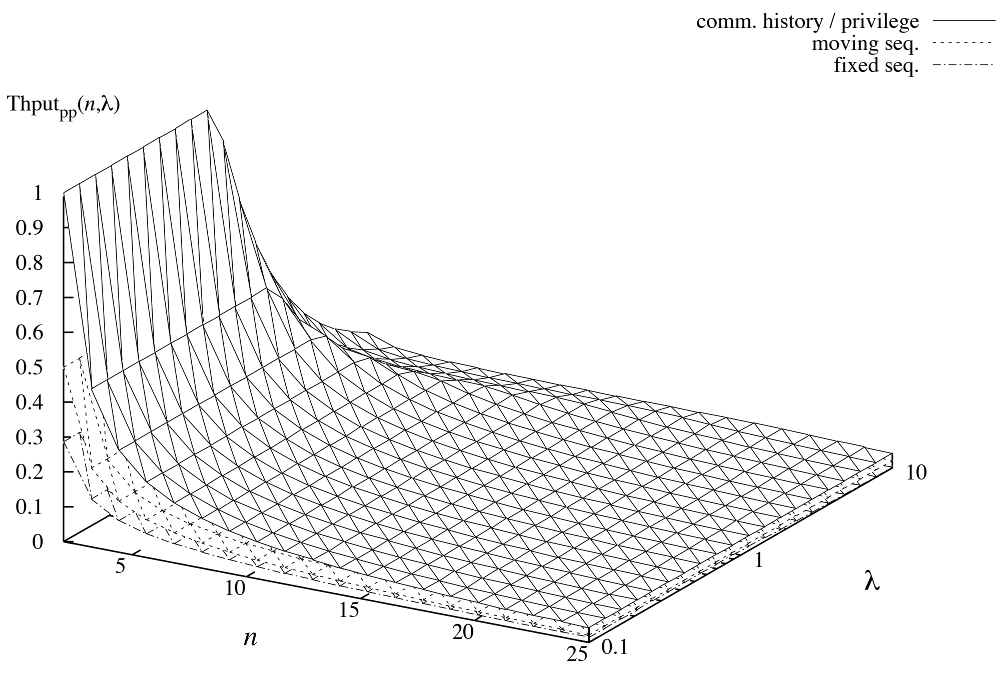
The throughput of the communication history and the privilege-based algorithms are identical. This is easily explained by the fact that both algorithms have exactly the same communication pattern under high load. Since both algorithms yield the same result, we plot them together in Figure 6.11 and Figure 6.12.

Throughput of Uniform Algorithms in Point-to-Point Networks

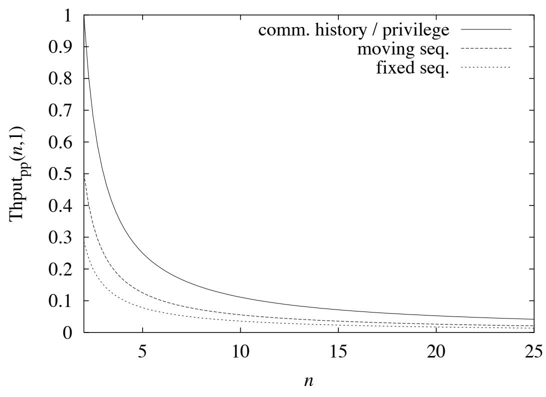
Figure 6.11 is split into three parts. Figure 6.11(a) represents the throughput of the algorithms in a two-dimensional parameter space defined by n and λ. Figure 6.11(b) show the evolution of the throughput when λ is fixed and n varies. Figure 6.11(c) cuts the surfaces of Figure 6.11(a) along the plane given by $n = 10$.

As seen from Figure 6.11, the communication history and the privilege-based algorithms can achieve the best throughput of the uniform algorithms. The communication history algorithms does not generate empty messages under a high and evenly distributed load. Consequently, the algorithm effectively generates only one single send to all for each application message. Similarly, the privilege-based algorithm generates the same amount of traffic because the high load makes it possible to piggyback all token messages.

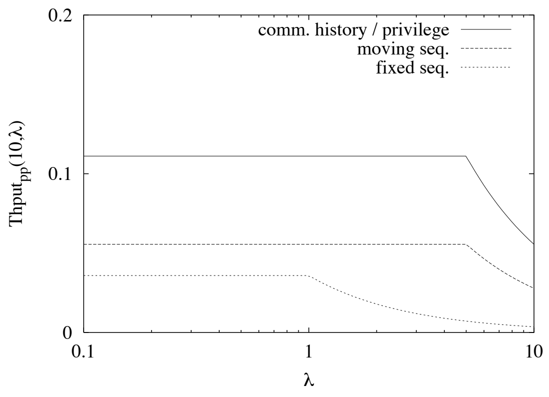
Figure 6.11 also shows that the two sequencer algorithms have a bad throughput. It is interesting to note that the moving sequencer algorithm always performs slightly better than the fixed sequencer one. This is easily explained by the fact that the moving sequencer uses a token-passing mechanism for the stabilization of messages. It turns out that token messages can easily benefit from the moving sequencer and be piggybacked on other messages. In contrast, the centralized acknowledgement scheme used by the fixed sequencer makes it more difficult to reduce its overhead



(a) Uniform algorithms ($\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$)



(b) $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda = 1)$



(c) $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n = 10, \lambda)$

Figure 6.11: Graphical representation of $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$ for uniform algorithms

in this way.

Figure 6.11(c) shows that the throughput of all uniform algorithms drops when λ increases over a certain threshold. This threshold depends however on the algorithm: when $n = 10$ (Fig.6.11(c)), the throughput of the fixed sequencer algorithm starts to drop when λ is greater than 1, while the throughput begins to decrease when λ is greater than 5 for the three other algorithms. This shows that the fixed sequencer is more sensitive to a lack of CPU resources than the other algorithms.

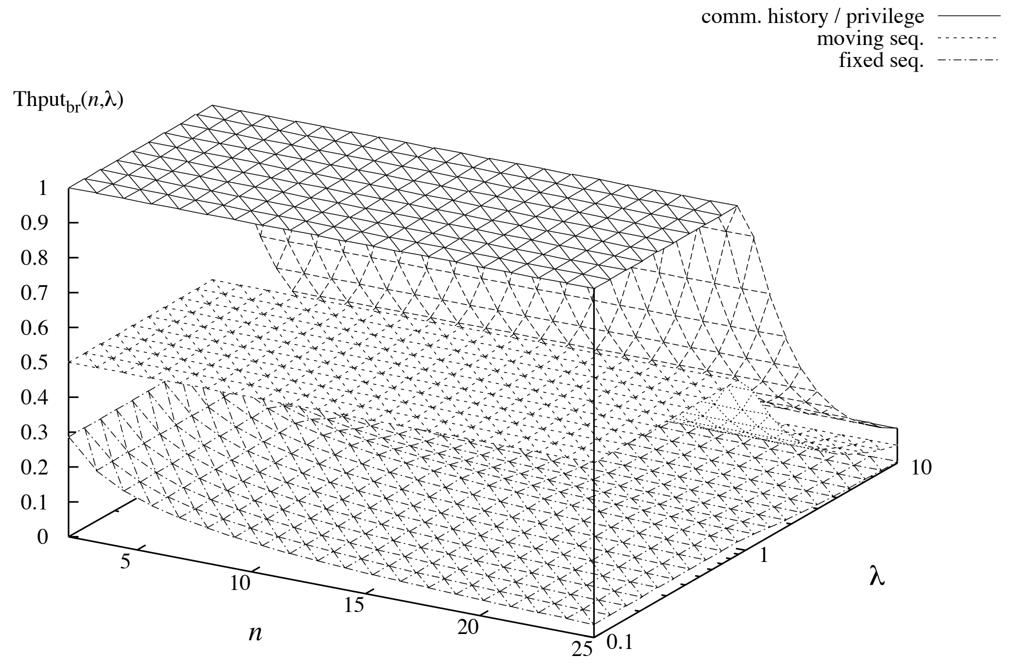
Although based on a similar approach, it is interesting to note that the moving sequencer and the fixed sequencer algorithm do not behave in the same way when λ increases. Indeed, as the parameter λ increases, the throughput of the moving sequencer algorithm starts to drop later than the throughput of the fixed sequencer algorithm (see Fig.6.11(c)). This is due to the fact that the moving sequencer algorithm distributes the load of sequencing messages evenly among all processes. On the contrary, the fixed sequencer algorithm concentrates this load on a single process (the sequencer). So, the CPU of the sequencer becomes a limiting factor when the value of λ increases beyond 1.

Throughput of Uniform Algorithms in Broadcast Networks

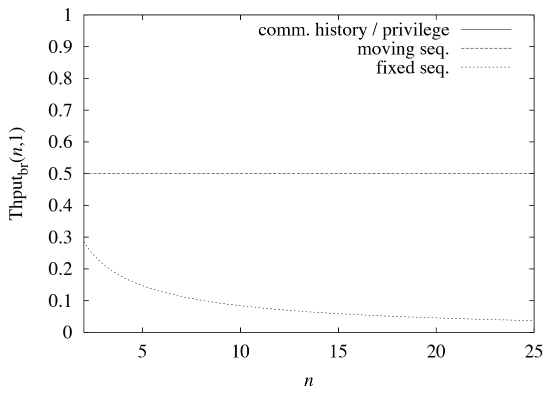
We now analyze the throughput of the uniform algorithms in a broadcast network. The results of $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$ are plotted on Figure 6.12, which is also split into three parts: the three-dimensional representation of the throughput, as a function of n and λ (Fig.6.12(a)); the projection on the n-axis in Figure 6.12(b); and the projection on the λ-axis in Figure 6.12(c). Note that the curve for the communication history (and privilege-based) algorithm is not visible on Figure 6.12(b) because its throughput is 100% for any value of n (see Fig.6.12(a)).

Figure 6.12 shows that the relative performance of the algorithms is the same than in point-to-point networks (compare with Fig.6.11). The algorithms however behave quite differently. For instance, except for the fixed sequencer algorithm, the throughput does not depend on the number of processes (see Fig.6.12(b)). This is a clear evidence that the communication history, the privilege-based, and the moving sequencer algorithms make a better use of the broadcast medium than the fixed sequencer algorithm. The latter algorithm is obviously penalized by its positive acknowledgement scheme used for the stabilization of messages.

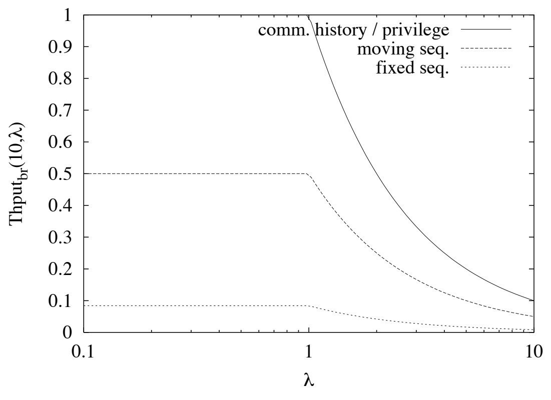
A comparison between Figure 6.12(c) and Figure 6.11(c) allows for a second observation. In both cases, the throughput of the algorithms decreases when λ goes beyond a certain threshold. Interestingly, in broadcast networks (Fig.6.12(c)), the throughput of *all* algorithms drops when λ increases beyond 1. This for instance reveals that the load balancing of the moving sequencer algorithm is of little help in a broadcast network. The actual weakness of the fixed sequencer algorithm is its stabilization mechanism which leaves only few opportunities for piggybacking messages.



(a) Uniform algorithms ($\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$)



(b) $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda=1)$



(c) $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n=10,\lambda)$

Figure 6.12: Graphical representation of $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$ for uniform algorithms

6.4.2 Throughput of Non-Uniform Algorithms

We now analyze the throughput of the non-uniform algorithms: *non-uniform privilege-based*, *non-uniform moving sequencer*, *non-uniform fixed sequencer*, and *destinations agreement*. As for the uniform algorithms, we first consider them in a point-to-point network, and then in a broadcast network. Figure 6.13 illustrates the results of $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$, and Figure 6.15 illustrates those obtained with $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$.

Throughput of Non-Uniform Algorithms in Point-to-Point Networks

The throughput of the non-uniform algorithms in point-to-point networks is illustrated on Figure 6.13. As usual, the figure is split into three parts: Figure 6.13(a) gives a three-dimensional view of the throughput, according to the two parameters n and λ; Figure 6.13(b) and Figure 6.13(c) show the curves when respectively λ and n are fixed. In addition, Figure 6.14 is a detailed view of Figure 6.13(a), where both the privilege-based and the destinations agreement algorithms have been removed. This makes it possible to see the intersection of the two surfaces that represent the fixed and the moving sequencer algorithms.

The privilege-based algorithm has the best throughput, as illustrated on Figure 6.13. Also, the moving sequencer algorithm has a better throughput than the destinations agreement algorithm. In spite of this, all three algorithms have a comparable behavior as the values of the parameters change.

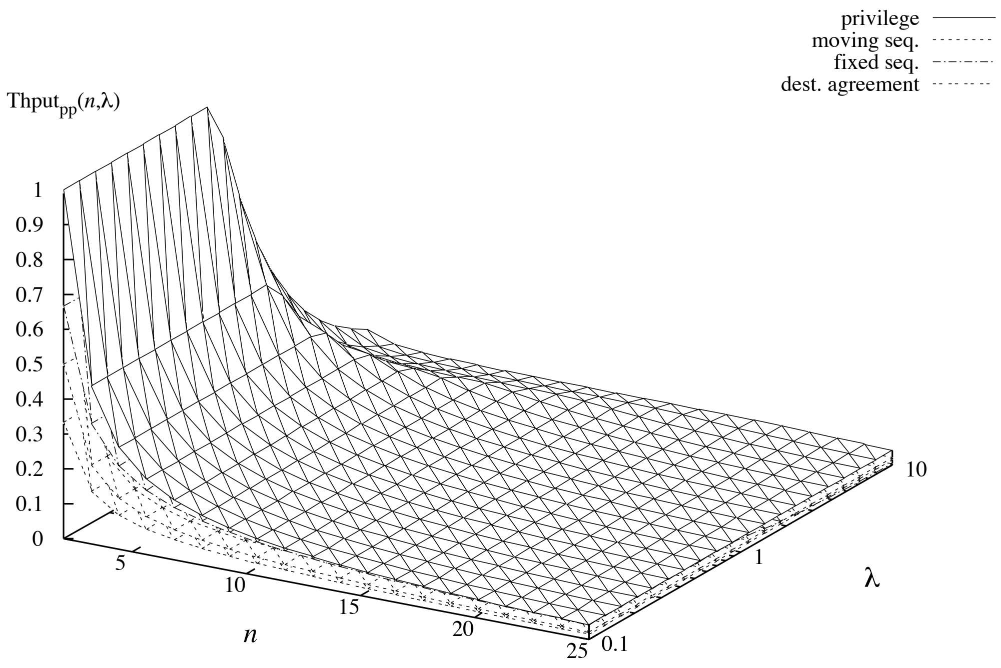
The behavior of the fixed sequencer algorithm is quite interesting. Indeed, its performance is largely dependent on the value of λ: the algorithm can move from the second position when λ is small, to the last position when λ is big. This is shown in Figure 6.13(c), but it is even more obvious on Figure 6.14. The fixed sequencer has a good throughput when λ is small. However, the sequencer comes quickly as a limiting factor when λ increases. The moving sequencer and the destinations agreement algorithms are less subject to this problem because they better distribute the load among the processes.

Throughput of Non-Uniform Algorithms in Broadcast Networks

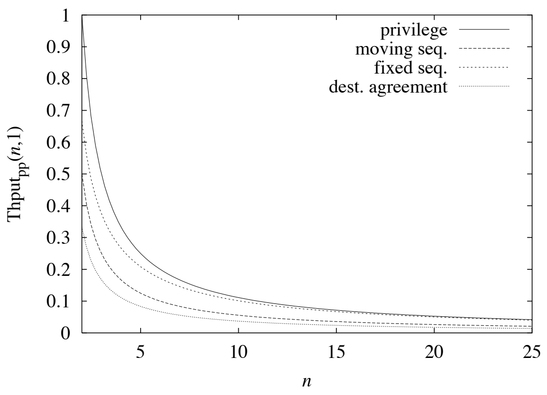
Figure 6.15 depicts the throughput of the non-uniform algorithms in a broadcast network. Figure 6.15(a) gives the overview of the measures, while Figure 6.15(b) and 6.15(c) show the evolution of $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$ when n and λ vary, respectively.

Figure 6.15 shows that the order between the four algorithms is well defined. The privilege-based algorithm has the best throughput, and the destinations agreement algorithm has the worst one. The fixed sequencer algorithm takes the second position, while the moving sequencer one has a slightly lower throughput and takes the third position.

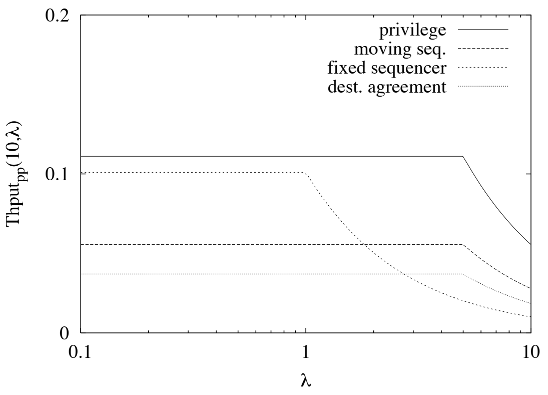
The two sequencer algorithms (moving and fixed) perform similarly, especially when n grows (see Fig.6.15(b)). The throughput of the fixed sequencer decreases when n increases, because of the first message sent by the algorithm. Indeed, in the fixed sequencer algorithm, the algorithm does not send the first message when the sender happens to be the sequencer. This situation reduces



(a) Non-uniform algorithms ($\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$)



(b) $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda = 1)$



(c) $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n = 10, \lambda)$

Figure 6.13: Graphical representation of $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$ for non-uniform algorithms

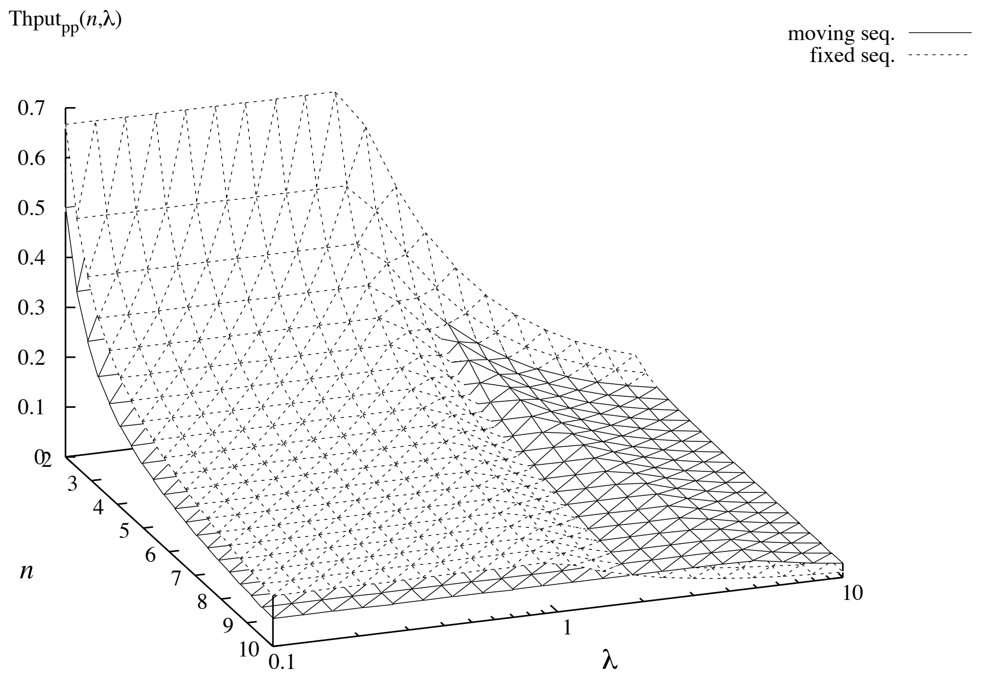


Figure 6.14: Graphical representation of non-uniform moving and fixed sequencer algorithms (detail of Figure 6.13(a))

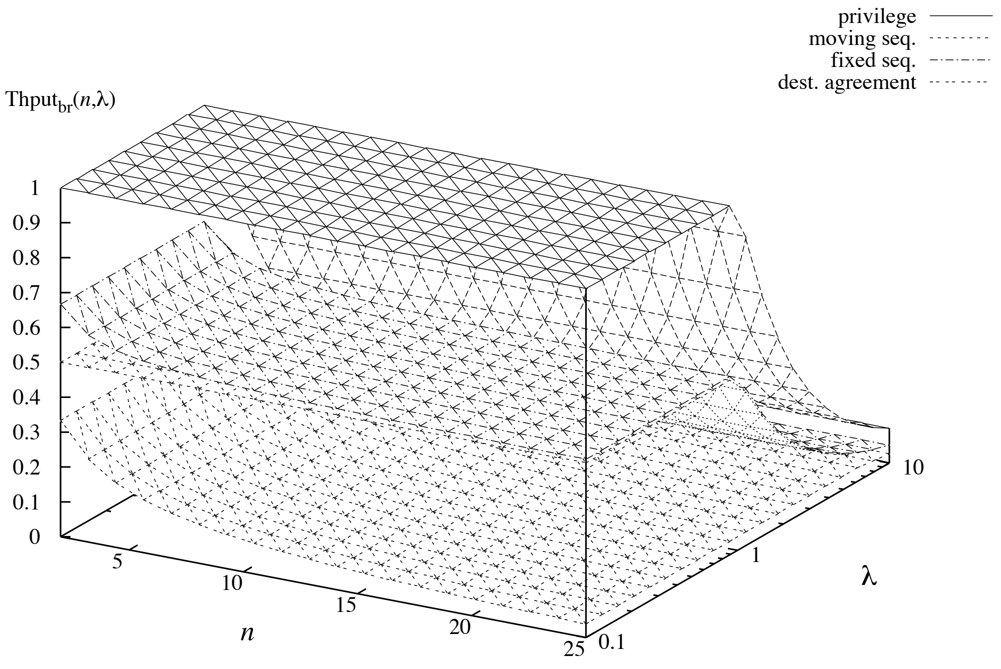
resource usage, but occurs less frequently as n increases. This explains why the fixed sequencer algorithm asymptotically (in terms of n) behaves like the moving sequencer algorithm.

Figure 6.15(b) leads to a second observation. Unlike the other three algorithms, the destinations agreement algorithm asymptotically tends to a null throughput as n grows, despite the broadcast network. This lack of scalability is due to the local timestamps that the sender has to gather from all destination processes. Broadcast network or not, this generates $O(n)$ messages.

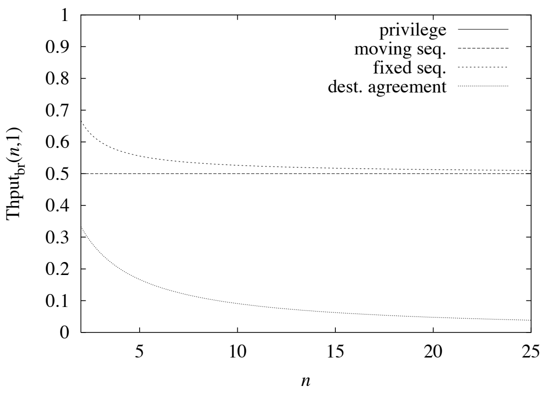
On the positive side, the destinations agreement algorithm is less sensitive to large values of λ than the other three algorithms. This is illustrated in Figure 6.15(c), where the throughput starts to drop when λ is larger than 3 (instead of $\lambda > 1$ for the other algorithms).

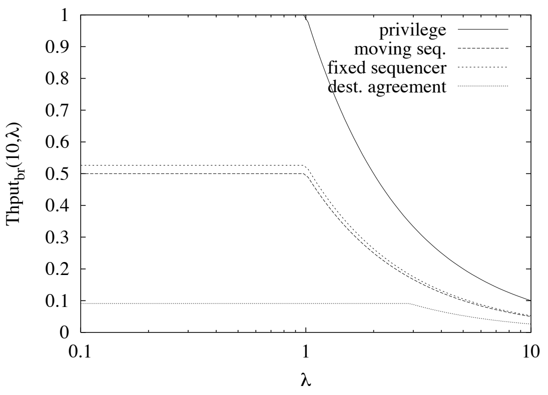
6.5 Summary

We have analyzed four uniform algorithms and four non-uniform algorithms for total order broadcast. Each algorithm represents one of the classes of total order broadcast algorithms as defined in Chapter 4. The analysis is based on the contention-aware metrics presented in Chapter 5. The results obtained through these metrics are significantly more relevant than those obtained with more conventional metrics: time and message complexity. The contention-aware metrics and the



(a) Non-uniform algorithms ($\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$)





(b) $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n, \lambda = 1)$

(c) $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n = 10, \lambda)$

Figure 6.15: Graphical representation of $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$ for non-uniform algorithms

conventional metrics also sometimes give opposite predictions.

Uniform algorithms The analysis of the uniform algorithms gives interesting results. First, the communication history algorithm has the best throughput, but also the best latency in a broadcast network. It also has the best latency in a point-to-point network, if λ is big. On the other hand, this algorithm is poorly scalable in a point-to-point network, with a quadratic degradation of performance as the number of processes increases. We should also point out that the method used to evaluate the throughput is particularly favorable for communication history and privilege-based algorithms.[5]

The privilege-based algorithm has the best throughput (identical to communication history algorithm). However, this algorithm also has the worst latency, except in a point-to-point network and only if the network is more important than the CPU (i.e., if λ is small).

The moving sequencer algorithm can be seen as a compromise between throughput and latency. Indeed, the performance of the algorithm is average in all cases. A positive thinking would say that it is never the worst choice, but one could also rightfully claim that it is never the best choice either!

Generally, the fixed sequencer algorithm has a good latency which, however, comes at the price of a low throughput. This is already expected from the results of the conventional metrics.

Non-uniform algorithms With non-uniform algorithms, the best overall results are achieved by the fixed sequencer algorithm. Indeed, this algorithm has the best latency and only comes in second position for the throughput. It is interesting to put these observations in perspective with those obtained for the uniform version of the algorithm. This indeed yields that uniformity comes at a very high price for this particular algorithm.

The privilege-based algorithm can achieve a very high throughput. This however comes at the price of latency, for which the algorithm has the third position. The privilege-based algorithm even takes the fourth position in a broadcast network, when the number of processes is large and the importance of the CPU and network resources is roughly equivalent (i.e., λ close to 1).

As for its uniform counterpart, the performance of the moving sequencer algorithm is generally average. It nevertheless has the best latency in a broadcast network, identical to the latency of the fixed sequencer algorithm.

Finally, the destinations agreement algorithm is the worst both in latency and throughput. It should however be noted that the destinations agreement algorithm can easily be transformed into a uniform algorithm, with no significant performance degradation. Inferring from the results obtained in this chapter, the resulting uniform algorithm would probably have a latency similar to the uniform fixed sequencer algorithm, and a slightly better throughput.

Tradeoffs Putting the results together yields that the choice of an appropriate algorithm depends on the requirements of the system in which it is to be used. Indeed, if a good throughput is more

[5]The method used to determine the throughput assumes a high load that is evenly distributed among all processes. In such a case, the communication history algorithm does not need to generate any empty message, and the privilege-based algorithm can piggyback all token messages on other messages.

important than a good latency, then either a communication history or a privilege-based algorithm is probably a good choice. Conversely, if latency is more important than throughput, then a fixed sequencer is more likely to meet the requirements. When both latency and throughput are equally important, a moving sequencer algorithm can provide a good compromise.

Of course, there are many other aspects to consider than performance only. The architectural properties of algorithms, or the semantics of the algorithm in case of failures (e.g., uniformity) are indeed often more important than just performance. For instance, if a system requires an open group architecture and a uniform behavior, then a uniform destinations agreement algorithm may have a good potential.

Chapter 7

Conclusion

Everything changes, nothing remains constant.

— **Siddharta Gautama Buddha** (ca. 563–483 B.C.)

7.1 Research Assessment

This research has led to four major contributions. The first major contribution is the definition of the semi-passive replication strategy. This variant of the passive replication technique can be implemented in asynchronous systems, without relying on any form of process controlled crash. The second important contribution is the classification and the survey of Total Order Broadcast algorithms. The third contribution is the definition of a set of metrics for distributed algorithms. The last major contribution is the comparison of the performance of each class of Total Order Broadcast algorithms.

Semi-passive replication This dissertation presents semi-passive replication, which is a passive replication scheme based on a variant of Consensus (Lazy Consensus). Expressing semi-passive replication in terms of a Consensus problem emphasizes the relation between passive replication and Consensus, but also between passive and active replication.

The main contribution of semi-passive replication is to show that a passive replication scheme can be implemented without relying on any form of process controlled crash. The first consequence is that it is then possible to implement replicated services using a passive replication scheme, with a better reaction time in case of failures. A second consequence is that it avoids the risk of removing correct processes from the group of replicas; a problem that is inherent to process controlled crash.

Other contributions can be mentioned. First, we give a specification of the Replication problem, expressed as an agreement problem. The specification encompasses both active and passive replication, as well as all hybrid replication schemes that we know about (i.e., semi-active replication, coordinator-cohort). Second, from the standpoint of the client, active replication and semi-passive replication follow exactly the same protocol. Combined with the fact that both can be implemented based on consensus, makes it much easier for both techniques to coexist.

Classification of Total Order Multicast algorithms Despite the large number of total order broadcast algorithms in the literature, the number of fundamentally different solutions is quite limited. This dissertation surveys about fifty different algorithms and tries to highlight their similarities rather than their differences.

The main contribution is the definition of five classes of total order broadcast algorithms, as well as the survey of existing algorithms. This provides a basis for further studies of total order broadcast, a set of criteria to describe new solutions to the problem, as well as a better understanding of the strengths and weaknesses of the different classes of algorithms.

Metrics for distributed algorithms This dissertation presents two metrics to analyze the performance of distributed algorithms: a latency metric and a throughput metric. These metrics are specifically adapted to evaluate distributed algorithms as they largely account for resource contention. This dissertation clearly shows that our contention-aware metrics can yield results that are far more detailed than those obtained with conventional metrics. Our metrics thus reveal a gap between complexity measures and performance evaluation techniques (e.g., simulation, performance measurements): they stand as a valid substitute to the former, and stand as a good complement to the latter.

Evaluation of Total Order Multicast algorithms This dissertation presents a comparative analysis of four uniform algorithms and four non-uniform algorithms for total order broadcast. The analysis is based both on the classification of total order broadcast algorithm and the contention-aware metrics. The results obtained through these metrics are significantly more relevant than results obtained with time or message complexity.

The results of this comparison yield unexpected results. For instance, communication history algorithms have a surprisingly good latency if used in a broadcast network. Also, the usual claim that a moving sequencer algorithm is superior to a fixed sequencer one because of load balancing seems to be quite an overstatement.

7.2 Open Questions & Future Research Directions

Research is mostly about looking for questions rather than answers. Indeed, finding the answers to important questions usually leads to asking further questions. This research is no exception, and we now present some of the related open questions and future directions.

Semi-passive replication in other models The semi-passive replication and the Lazy Consensus algorithm presented in Chapter 3 consider a model with benign failures, in which processes can crash but never recover. An interesting question is how these results could be adapted to more complex models: systems where processes can crash and recover, systems where correct processes can be disconnected because of network partitions, or systems with Byzantine processes.

Adaptive failure detectors for Lazy Consensus Semi-passive replication and Lazy Consensus make it possible to use failure detectors that are considerably more aggressive than those used in most group communication systems. Sergent [Ser98] has shown, in the context of Consensus, that an ideal timeout value for a failure detector is in the order of the execution a single round in the Consensus, that is, a few millisecond on a LAN. This value is the smallest value for which the response time of the algorithm does not increase in a failure-free run, and has been found by simulating various executions of the algorithm. A simulation however fails to account for the extreme unpredictability of general-purpose LANs [UDS00b].

The ideal timeout value varies over time, and an ideal failure detector should match this evolution. The definition of an adaptive failure detector that approximates this evolution may be a strong asset for group communication system based on aggressive failure detectors.

Dissection of Total Order Broadcast algorithms The classification and the evaluation of total order broadcast algorithms has taught us that, despite the number of different algorithms, they use a very limited number of basic communication patterns. This particularly obvious in Figure A.9 (p.156) that depicts the communication pattern of every algorithm that is considered in the evaluation of Chapter 6. For instance, it is possible to change the centralized communication scheme of the destinations agreement representative (based on Skeen's algorithm) into a decentralized one (see Fig.4.10, p.80). As a result, the communication pattern of the destinations agreement algorithm becomes very similar to that of the communication history one (based on Lamport's algorithm), even though the actual actions of either algorithms are fundamentally different.

A dissection of total order broadcast algorithms into basic building blocks could bring many benefits. Aside from a better understanding of the algorithms, this would probably make it possible to *quantify* the cost of each property and guarantee that a total order broadcast algorithm can enforce (e.g., uniformity). In other words, this would give a better understanding of the tradeoffs that exist between specification and performance.

Analysis of failure management mechanisms The analysis presented in this dissertation can be extended to compare failure management mechanisms (e.g., group membership, rotating coordinator). This however raises an important question: "is it possible to consider the failure management mechanism independently from any specific algorithm?" A positive answer to this question would make it considerably easier to assess the exact cost of specific algorithms in the case of failure.

Intuitively, this seems quite difficult. Indeed, it seems that the failure management mechanism could *conceptually* be separated from the algorithm. However, this poses some problems when it comes to the proper *tuning* of the failure management mechanisms. In this case, it often turns out that the failure detection mechanism is so mingled with the algorithm that is becomes impossible to consider the former in isolation of the latter.

Metrics for distributed algorithms The contention-aware metrics presented in this dissertation are very promising. But, their definition is not sufficient to analyze and compare algorithms in the context of failures. To overcome this limitation, it is important to extend the definition of the metrics in order to integrate the evaluation of performance when failures occur.

The metrics can be extended in various ways. First, it is important to define several relevant scenarios that measure, for instance, the worst latency in the case of a crash, or the additional cost of incorrect suspicions in terms of throughput and latency.

Second, as failure mechanisms are usually based on some timeout mechanism, it may be necessary to extend our metrics with an adequate timing model. Also, it is necessary to determine "realistic" timeout values for a given setup; consisting of a system model, an algorithm, a problem, and a failure management mechanism. More specifically, given two algorithms, the respective optimal values for the timeout are likely to be different. It is then necessary to determine those values in order to compare the two algorithms.

Total Order Broadcast for database replication The use of Total Order Broadcast to improve database replication protocols recently took quite some attention [WPS+00, WPS99, PGS98, AAEAS97]. According to the simulation results published so far, the approach seems very promising. Nevertheless, Total Order Broadcast algorithms have a long-lasting reputation of poor performance in the community of database system researchers.

A deeper analysis of existing database replication schemes [WPS+00, WPS99], combined with the results obtained through our analysis of Total Order Broadcast algorithms, could probably fight off the misconception that Total Order Broadcast is a source of poor performance. For instance, the combination of both work could make it possible to design Total Order Broadcast algorithms specifically adapted to the problem of database replication. This could provide a direction to improve on the work of Pedone and Schiper [PS98, PS99].

– 終 –

Bibliography

[AADW94] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In S. Goldwasser, editor, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 401–411, Los Alamitos, CA, USA, November 1994.

[AAEAS97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, number 1300 in Lecture Notes in Computer Science, pages 496–503, Passau, Germany, August 1997. Extended abstract.

[ACM95] G. Alvarez, F. Cristian, and S. Mishra. On-demand asynchronous atomic broadcast. In *5th IFIP Working Conference on Dependable Computing for Critical Applications*, Urbana-Champaign, IL, USA, September 1995.

[ACT97] M. K. Aguilera, W. Chen, and S. Toueg. Quiescent reliable communication and quiescent consensus in partitionable networks. TR 97-1632, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, June 1997.

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

[ACT99] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999. Special issue on distributed algorithms.

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, San Francisco, CA, USA, 1976.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 76–84, Boston, MA, USA, July 1992.

[ADLS94] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeier. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, January 1994.

[AKP92] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 571–580, Victoria, BC, Canada, May 1992.

[AM92] E. Anceaume and P. Minet. Étude de protocoles de diffusion atomique. TR 1774, INRIA, Rocquencourt, France, October 1992.

[AMMS+93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. 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 (ICDCS-13)*, pages 551–560, Pittsburgh, PA, USA, May 1993.

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[AMMSB98] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, May 1998.

[Anc93a] E. Anceaume. *Algorithmique de Fiabilisation de Systèmes Répartis*. PhD thesis, Université de Paris-sud (Paris XI), Paris, France, 1993.

[Anc93b] E. Anceaume. A comparison of fault-tolerant atomic broadcast protocols. In *Proceedings of the 4th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-4)*, pages 166–172, Lisbon, Portugal, September 1993.

[Anc97] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, pages 292–301, Seattle, WA, USA, June 1997.

[AS98] Y. Amir and J. Stanton. The Spread wide area group communication system. TR CDNS-98-4, John Hopkins University, Baltimore, MD, USA, 1998.

[AW95] J. Aspnes and O. Waarts. A modular measure of competitiveness for distributed algorithms (abstract). In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC-14)*, page 252, Ottawa, Ontario, Canada, August 1995.

[AW96] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proceedings of the 28th ACM Symposium on Theory of Computing (STOC)*, pages 237–246, Philadelphia, PA, USA, May 1996.

[BB93] P. Berman and A. A. Bharali. Quick atomic broadcast. In *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG-7)*, number 725 in Lecture Notes in Computer Science, pages 189–203, Lausanne, Switzerland, September 1993. Extended abstract.

[BCG91] K. Birman, R. Cooper, and B. Gleeson. Programming with process groups: Group and multicast semantics. Technical Report TR91-1185, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, January 1991.

[BFR92] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 39–50, Victoria, BC, Canada, May 1992.

[BHSC98] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[Bir94] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *ACM Operating Systems Review*, 28(1):11–21, January 1994.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BJREA85] K. P. Birman, T. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):502–508, June 1985.

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

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

[Bud93] N. Budhiraja. *The Primary-Backup Approach: Lower and Upper Bounds*. PhD thesis, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, June 1993. 93/1353.

[BVR93] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[Car85] R. Carr. The Tandem global update protocol. *Tandem Systems Review*, June 1985.

[CASD84] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From simple message diffusion to Byzantine agreement. Technical Report 4540, IBM Research Lab., San Jose, CA, USA, December 1984.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, Ann Arbor, Michigan, USA, June 1985.

[CASD95] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[CBPD99] B. Charron-Bost, F. Pedone, and X. Défago. Private communications. Showed an example illustrating the fact that even the combination of strong agreement, strong total order, and strong integrity does not prevent a faulty process from reaching an inconsistent state., November 1999.

[CD89] B. Chor and C. Dwork. Randomization in Byzantine Agreement. *Adv. Comput. Res.*, 5:443–497, 1989.

[CdBM94] F. Cristian, R. de Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering*, 1(4):177–201, June 1994.

[CDSA90] F. Cristian, D. Dolev, R. Strong, and H. Aghili. Atomic broadcast in a real-time environment. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, number 448 in Lecture Notes in Computer Science, pages 51–71. Springer-Verlag, 1990.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. on Parallel & Distributed Systems*, 10(6):642–657, June 1999.

[CFM87] M. Cart, J. Ferrié, and S. Mardyanto. Atomic broadcast protocol, preserving concurrency for an unreliable broadcast network. In J. Cabanel, G. Pujole, and A. Danthine, editors, *Proc. IFIP Conf. on Local Communication Systems: LAN and PBX*. Elsevier Science Publishers, 1987.

[CH98] G.-M. Chiu and C.-M. Hsiao. A note on total ordering multicast using propagation trees. *IEEE Trans. on Parallel and Distributed Systems*, 9(2):217–223, February 1998.

[CHD98] G. V. Chockler, N. Huleihel, and D. Dolev. An adaptive total ordered multicast protocol that tolerates partitions. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17)*, pages 237–246, Puerto Vallarta, Mexico, June 1998.

[Che00] W. Chen. *On the Quality of Service of Failure Detectors*. PhD thesis, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, 2000. *To appear*. Draft available online at http://www.cs.cornell.edu/Info/People/weichen/research/mypapers/thesis.ps.

[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.

[CKP$^+$96] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[CL96] J. Córdova and Y.-H. Lee. Multicast trees to provide message ordering in mesh networks. *Computer Systems Science & Engineering*, 11(1):3–13, January 1996.

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

[CM95] F. Cristian and S. Mishra. The pinwheel asynchronous atomic broadcast protocols. In *2nd Int'l Symposium on Autonomous Decentralized Systems*, pages 215–221, Phoenix, AZ, USA, April 1995.

[CMA97] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distributed System Engineering Journal*, 4(2):109–128, June 1997.

[CMMS96] X. Chen, L. E. Moser, and P. M. Melliar-Smith. Reservation-based totally ordered multicasting. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 511–519, Hong Kong, May 1996.

[Cri90] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Real-Time Systems*, 2(3):195–212, September 1990.

[Cri91] F. Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, 1991.

[CS93] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symp. on Operating Systems Principles (SoSP-14)*, pages 44–57, Asheville, NC, USA, December 1993.

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

[CTA00] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (ICDSN/FTCS-30)*, New York, NY, USA, June 2000.

[Das92] M. Dasser. TOMP: A total ordering multicast protocol. *ACM Operating Systems Review*, 26(1):32–40, January 1992.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[Dee89] S. E. Deering. RFC 1112: Host extensions for IP multicasting, August 1989.

[DFS99] X. Défago, P. Felber, and A. Schiper. Optimization techniques for replicating CORBA objects. In *Proceedings of the 4th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'99)*, Santa Barbara, CA, USA, January 1999.

[DGF99] C. Delporte-Gallet and H. Fauconnier. Real-time fault-tolerant atomic broadcast. In *Proceedings of the 18th Symposium on Reliable Distributed Systems (SRDS-18)*, pages 48–55, Lausanne, Switzerland, October 1999.

[DHW97] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, November 1997.

[DKM93] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environments. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 544–553, Toulouse, France, June 1993.

[DKV83] P. Decitre, A. Khider, and G. Vandôme. Protocole de diffusion fiable pour réseaux locaux. Technical Report RR-347, Laboratoire IMAG, Grenoble, France, January 1983.

[DLS88] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

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

[DMS98] X. Défago, K. R. Mazouni, and A. Schiper. Highly available trading system: Experiments with CORBA. In N. Davies, K. Raymond, and J. Seitz, editors, *Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 91–104, The Lake District, UK, September 1998. Springer-Verlag.

[DS00] X. Défago and A. Schiper. Semi-passive replication and Lazy Consensus. Technical Report DSC/2000/027, École Polytechnique Fédérale de Lausanne, Département de Systèmes de Communication, Lausanne, Switzerland, May 2000. *Submitted for publication.*

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

[DSU00] X. Défago, A. Schiper, and P. Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Technical report, École Polytechnique Fédérale de Lausanne, Département de Systèmes de Communication, Lausanne, Switzerland, 2000. *To appear.*

[EAP99] D. Essamé, J. Arlat, and D. Powell. PADRE: a protocol for asymmetric duplex redundancy. In *IFIP 7th Working Conference on Dependable Computing in Critical Applications (DCCA-7)*, pages 213–232, San Jose, CA, USA, January 1999.

[EMS95] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS-15)*, pages 296–306, Vancouver, Canada, May 1995.

[End99] M. Endler. An atomic multicast protocol for mobile computing. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL M-3)*, pages 56–63, 1999.

[FDES99] P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA objects: a marriage between active and passive replication. In *Second IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, Helsinki, Finland, June 1999.

[FIMR98] U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS-17)*, pages 228–234, West Lafayette, IN, USA, October 1998.

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

[FM90] A. Freier and K. Marzullo. MTP: An atomic multicast transport protocol. Technical Report TR90-1141, Cornell University, Computer Science Department, July 1990.

[FVR97] R. Friedman and R. Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE Symposium on High Performance Distributed Computing*, pages 233–242, Portland, OR, USA, August 1997.

[GHP83] C. Gailliard and D. Hué-Petillat. Un protocole de diffusion atomique. *TSI*, 2(6), December 1983.

[GMR98] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science*, 196(1–2):3–29, 6 April 1998.

[GMS89] H. Garcia-Molina and A. Spauster. Message ordering in a multicast environment. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS-9)*, pages 354–361, Newport Beach, CA, USA, June 1989.

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

[Gra79] J. N. Gray. Notes on data base operating systems. In *Operating Systems: an advanced course*, pages 393–481. Springer-Verlag, 1979.

[GS93] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math.* Texts and monographs in computer science. Springer-Verlag, 1993.

[GS95a] R. Guerraoui and A. Schiper. A generic multicast primitive to support transactions on replicated objects in distributed systems. In *Proceedings of the 5th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-5)*, pages 334–342, Cheju Island, Korea, August 1995.

[GS95b] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 121–132. Springer-Verlag, September 1995.

[GS97a] R. Guerraoui and A. Schiper. Genuine atomic multicast. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG-11)*, number 1320 in Lecture Notes in Computer Science, pages 141–154, Saarbrücken, Germany, September 1997.

[GS97b] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[GS97c] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS-17)*, pages 578–585, Baltimore, Maryland, USA, May 1997.

[GT89] A. Gopal and S. Toueg. Reliable broadcast in synchronous and asynchronous environment. In *Proceedings of the 3rd International Workshop on Distributed Algorithms (WDAG-3)*, number 392 in Lecture Notes in Computer Science, pages 111–123, Nice, France, September 1989.

[GT91] A. Gopal and S. Toueg. Inconsistency and contamination. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC-10)*, pages 257–272, Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC-10), August 1991.

[HMR98] M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS)*, pages 280–286, West Lafayette, IN, USA, October 1998.

[HMRT99] M. Hurfin, R. Macêdo, M. Raynal, and F. Tronel. A general framework to solve agreement problems. In *Proceedings of the 18th Symposium on Reliable Distributed Systems (SRDS)*, pages 56–67, Lausanne, Switzerland, October 1999.

[HP97] A. Heddaya and K. Park. Congestion control for asynchronous parallel computing on workstation networks. *Parallel Computing*, 23(13):1855–1875, December 1997.

[HR97] M. Hurfin and M. Raynal. Fast asynchronous consensus with a weak failure detector. Technical Report PI-1118, IRISA, Rennes, France, July 1997. ftp://www.irisa.fr/techreports/1997.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, May 1994.

[HYF99] J.-H. Huang, C.-C. Yang, and N.-C. Fang. A novel congestion control mechanism for multicast real-time connections. *Computer Communications*, 22:56–72, 1999.

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

[JEG99] P. R. James, M. Endler, and M.-C. Gaudel. Development of an atomic-broadcast protocol using LOTOS. *Software—Practice and Experience*, 29(8):699–719, 1999.

[Jia95] X. Jia. A total ordering multicast protocol using propagation trees. *IEEE Trans. on Parallel and Distributed Systems*, 6(6):617–627, June 1995.

[KD96] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC-15)*, pages 68–76, Philadelphia, PA, USA, May 1996.

[KD00] I. Keidar and D. Dolev. Totally ordered broadcast in the face of network partitions. In D. Avresky, editor, *Dependable Network Computing*, chapter 3, pages 51–75. Kluwer Academic Publications, January 2000.

[KDK$^+$89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, February 1989.

[KGR90] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avižienis and J.-C. Laprie, editors, *Dependable Computing for Critical Applications (DCCA)*, volume 4, pages 411–429, 1990.

[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-19)*, pages 424–431, Austin, TX, USA, June 1999.

[Kri96] E. V. Krishnamurthy. Complexity issues in parallel and distributed computing. In A. Y. H. Zomaya, editor, *Parallel & Distributed Computing Handbook*, pages 89–126. McGraw-Hill, 1996.

[Kro96] L. I. Kronsjö. PRAM models. In A. Y. H. Zomaya, editor, *Parallel & Distributed Computing Handbook*, pages 163–191. McGraw-Hill, 1996.

[KT91a] M. F. Kaashoek and A. S. Tanenbaum. Fault tolerance using group communication. *ACM Operating Systems Review*, 25(2):71–74, April 1991.

[KT91b] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, pages 222–230, Arlington, TX, USA, May 1991.

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

[Lam84] L. Lamport. Using time instead of time-outs in fault-tolerant systems. *ACM Transactions on Programming Languages and Systems*, 6(2):256–280, 1984.

[Lap92] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese*, volume 5 of *Dependable Computing and Fault Tolerant Systems*. Springer-Verlag, 1992.

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

[LLB91] G. Le Lann and G. Bres. Reliable atomic broadcast in distributed systems with omission faults. *ACM Operating Systems Review, SIGOPS*, 25(2):80–86, April 1991.

[LS76] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed storage system. Technical report, Computer Sciences Laboratory, Xerox Palo Alto Research Center, Palo Alto, CA, USA, 1976.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[LYZ91] C.-C. Lim, L.-J. Yao, and W. Zhao. A comparative study of three token ring protocols for real-time communications. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, pages 308–317, May 1991.

[MA91a] P. Minet and E. Anceaume. ABP: An atomic broadcast protocol. Technical Report 1473, INRIA, Rocquencourt, France, June 1991.

[MA91b] P. Minet and E. Anceaume. Atomic broadcast in one phase. *ACM Operating Systems Review*, 25(2):87–90, April 1991.

[Mal94] D. Malkhi. *Multicast Communication for High Availability*. PhD thesis, Hebrew University of Jerusalem, Israel, May 1994.

[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.

[May92] E. Mayer. An evaluation framework for multicast ordering protocols. In *Proceedings of the Conference on Applications, Technologies, Architecture, and Protocols for Computer Communication (SIGCOMM)*, pages 177–187, August 1992.

[MFSW95] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, TX, USA, October 1995.

[MMS95] L. E. Moser and P. M. Melliar-Smith. Total ordering algorithms for asynchronous Byzantine systems. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-9)*, number 972 in Lecture Notes in Computer Science, pages 242–256, Le Mont-St-Michel, France, September 1995.

[MMS99] L. E. Moser and P. M. Melliar-Smith. Byzantine-resistant total ordering algorithms. *Information and Computation*, 150(1):75–111, April 1999.

[MMSA91] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Total ordering algorithms. In *ACM Annual Computer Science Conference, Preparing for the 21st Century*, pages 375–380, 1991.

[MMSA93] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *SIAM Journal on Computing*, 22(4):727–750, August 1993.

[MMSA$^+$95] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulis, and T. P. Archambault. The Totem system. In *Proceedings of the 25rd International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 61–66, Pasadena, CA, USA, 1995.

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

[MPS93] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: a communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87–103, 1993.

[MR99] A. Mostéfaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, number 1693 in Lecture Notes in Computer Science, pages 49–63, Bratislava, Slovak Republic, September 1999.

[MSM89] P. M. Melliar-Smith and L. E. Moser. Fault-tolerant distributed systems based on broadcast communication. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS-9)*, pages 129–134, Newport Beach, CA, USA, June 1989.

[MSMA90] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[NCN88] S. Navaratnam, S. T. Chanson, and G. W. Neufeld. Reliable group communication in distributed systems. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS-8)*, pages 439–446, San Jose, CA, USA, June 1988.

[Ng91] T. P. Ng. Ordered broadcasts for large applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems (SRDS-10)*, pages 188–197, Pisa, Italy, September 1991.

[NT90] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[OGS97] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report 97/239, École Polytechnique Fédérale de Lausanne, Switzerland, August 1997.

[PBS89] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, 1989.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, number 1470 in Lecture Notes in Computer Science, pages 513–520, Southampton, UK, September 1998.

[Pol94] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994.

[Pow91] D. Powell. *Delta4: A Generic Architecture for Dependable Distributed Computing*, volume 1 of *ESPRIT Research Reports*. Springer-Verlag, 1991.

[PS98] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC-12)*, number 1499 in Lecture Notes in Computer Science, pages 318–332, Andros, Greece, September 1998.

[PS99] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC-13)*, number 1693 in Lecture Notes in Computer Science, pages 94–108, Bratislava, Slovak Republic, September 1999.

[RA81] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 4(1):9–17, January 1981.

[Ray88] M. Raynal. *Networks and Distributed Computation: concepts, tools, and algorithms*. MIT Press, 1988.

[Ray91] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *ACM Operating Systems Review*, 25(2):47–50, April 1991.

[Rei94a] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science, pages 99–110, Dagstuhl Castle, Germany, September 1994. Springer-Verlag.

[Rei94b] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS-2)*, pages 68–80, 1994.

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

[RGS98] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proc. of the 7th IEEE International Conference on Computer Communications and Networks*, pages 840–847, Lafayette (LA), USA, October 1998.

[RM89] B. Rajagopalan and P. McKinley. A token-based protocol for reliable, ordered multicast communication. In *Proceedings of the 8th Symposium on Reliable Distributed Systems (SRDS-8)*, pages 84–93, Seattle, WA, USA, October 1989.

[RV92] L. Rodrigues and P. Veríssimo. *x*AMP: a multi-primitive group communications service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems (SRDS-11)*, pages 112–121, Houston, TX, USA, October 1992.

[RVC93] L. Rodrigues, P. Veríssimo, and A. Casimiro. Using atomic broadcast to implement a posteriori agreement for clock synchronization. In *Proceedings of the 12th Symposium on Reliable Distributed Systems (SRDS-12)*, pages 115–124, Princeton, NJ, USA, 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. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 7, pages 169–198. Addison-Wesley, second edition, 1993.

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

[SDS99] N. Sergent, X. Défago, and A. Schiper. Failure detectors: Implementation issues and impact on consensus performance. Technical Report SSC/1999/019, École Polytechnique Fédérale de Lausanne, Switzerland, May 1999.

[Ser98] N. Sergent. *Soft Real-Time Analysis of Asynchronous Agreement Algorithms Using Petri Nets*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1808.

[SH97] S.-P. Shieh and F.-S. Ho. A comment on "a total ordering multicast protocol using propagation trees". *IEEE Trans. on Parallel and Distributed Systems*, 8(10):1084, October 1997.

[Shr94] S. K. Shrivastava. To CATOCS or not to CATOCS, that is the .. *ACM Operating Systems Review*, 28(4):11–14, October 1994.

[SM96] J. B. Sussman and K. Marzullo. Comparing primary-backup and state machines for crash failures. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC-15)*, page 90, Philadelphia, PA, USA, May 1996. Brief announcement.

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

[ST85] D. D. Sleator and R. E. Tarjan. Amortised efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[TBW95] K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, September 1995.

[TFC99] C. Toinard, G. Florin, and C. Carrez. A formal method to prove ordering properties of multicast algorithms. *ACM Operating Systems Review*, 33(4):75–89, October 1999.

[Tse89] L. C. N. Tseung. Guaranteed, reliable, secure broadcast networks. *IEEE Network Magazine*, 3(6):33–37, November 1989.

[UDS00a] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of Atomic Broadcast algorithms. In *Proc. of the 9th IEEE Conference on Computer Communication and Network (ICCCN)*, October 2000.

[UDS00b] P. Urbán, X. Défago, and A. Schiper. The FLP impossibility result: A pragmatic illustration. Technical report, École Polytechnique Fédérale de Lausanne, Département de Systèmes de Communication, Lausanne, Switzerland, 2000. *To appear*.

[Val90] L. G. Valiant. A bridging model for parallel architectures. *Communications of the ACM*, 33(8):103–111, August 1990.

[VRB89] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: A highly parallel atomic multicast protocol. *Computer Communication Review*, 19(4):83–93, September 1989. Proc. of the SIGCOMM'89 Symposium.

[VRBC93] R. Van Renesse, K. P. Birman, and R. Cooper. The HORUS system. Technical report, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, 1993.

[VRBG$^+$95] R. Van Renesse, K. P. Birman, B. B. Glade, K. Guo, et al. Horus: A flexible group communications system. TR 95-1500, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, March 1995.

[VRC97] P. Veríssimo, L. Rodrigues, and A. Casimiro. CesiumSpray: A precise and accurate global clock service for large-scale systems. *Real-Time Systems*, 12(3):243–294, May 1997.

[WMK94] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In Springer-Verlag, editor, *Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science, pages 33–57, Dagstuhl Castle, Germany, September 1994.

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

[WPS$^+$00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS-20)*, pages 264–274, Taipei, Taiwan, April 2000.

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

[YST94] C. Yahata, J. Sakai, and M. Takizawa. Generalization of consensus protocols. In *Proc. of the 9th International Conference on Information Networking (ICOIN-9)*, pages 419–424, Osaka, Japan, 1994.

[YT95] C. Yahata and M. Takizawa. General protocols for consensus in distributed systems. In *Proc. of the 6th Int'l Conf. on Database and Expert Systems Applications (DEXA'95)*, number 978 in Lecture Notes in Computer Science, pages 227–236, London, UK, September 1995. Springer-Verlag.

[ZH95] P. Zhou and J. Hooman. Formal specification and compositional verification of an atomic broadcast protocol. *Real-Time Systems*, 9(2):119–145, September 1995.

[ZJ98] H. Zou and F. Jahanian. Real-time primary-backup replications with temporal consistency. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 48–56, Amsterdam, The Netherlands, May 1998.

Appendix A

Representative Algorithms for Totally Ordered Broadcast

Assumptions:

- group membership (exact specification is not relevant).
- reliable communication channels (implies no partition).
- processes fail by crashing (and remain crashed forever).
- no bound on communication delay or process speed (asynchronous).
- access to local physical clocks (not synchronized).

For the sake of conciseness, the algorithms that are presented in this section are based on a slightly unusual definition of the primitive “send to all”. There are actually two policies regarding this primitive.

1. Unless stated otherwise, when a process p sends a message to all, it sends that message to all processes *except* itself.
2. When it is *explicitly* stated that a process p sends a message to all *including itself*, the copy of the message that is sent to itself is considered “virtual” and does not actually transit on the network.

In Algorithm A.1, we introduce an arbitrary delay Δ_{live} that is necessary for the liveness of the algorithm.

Uniform Communication History

$$
\begin{aligned}
&\mathrm{Latency}_{\mathrm{pp}}(\text{Alg. A.1})(n,\lambda)\\
&= \begin{cases}
4\lambda+2 & \text{if } n=2\\
6\lambda+2+\max(0,2-\lambda)+\max(0,1-\lambda)+\max(0,1-2\lambda) & \text{if } n=3\\
3(n-1)\lambda+1 & \text{if } n\le\lambda+2\\[1em]
\frac{n^2-3n}{2}+2n\lambda+\lfloor\lambda\rfloor\lambda-\frac{\lfloor\lambda+3\rfloor\cdot\lfloor\lambda\rfloor}{2} & \\
\quad+\left\{\begin{array}{ll} 1 & \text{if } \lfloor\lambda\rfloor=n-3\\ 0 & \text{otherwise}\end{array}\right\} & \text{if } n\le 2\lfloor\lambda\rfloor+3 \text{ and } \lambda\ge 3\\[1em]
\frac{n^2-n}{2}+2n\lambda+\lfloor\lambda\rfloor\lambda-\frac{\lfloor\lambda+3\rfloor\cdot\lfloor\lambda\rfloor}{2} & \\
\quad-\lfloor 2\lambda\rfloor-3+\left\{\begin{array}{ll} \lfloor 2\lambda-1\rfloor & \text{if } n=5\\ 2 & \text{if } n=\lfloor 2\lambda+1\rfloor\\ 1 & \text{if } n=7 \text{ and } \lambda=\frac{11}{4}\\ 0 & \text{otherwise}\end{array}\right\} & \text{if } n\le 4\lambda-4\\[1em]
n^2-n+\left\{\begin{array}{ll} 2\lambda & \text{if } \lambda<1\\ \left\{\begin{array}{ll} 5\lambda-3 & \text{if } n=4\\ 4\lambda-2 & \text{if } n>4\end{array}\right\} & \text{if } 1\le\lambda<2\\ 5\lambda-4 & \text{if } 2\le\lambda<3\\ 2\lfloor\lambda\rfloor\lambda-\lfloor\lambda\rfloor-\lfloor\lambda\rfloor^2+5 & \text{otherwise}\end{array}\right\} & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$\mathrm{Latency}_{\mathrm{br}}(\text{Alg. A.1})(n,\lambda)=2(2\lambda+1)+(n-2)\max(1,\lambda)$$

$$\mathrm{Thput}_{\mathrm{pp}}(\text{Alg. A.1})(n,\lambda)=\frac{1}{(n-1)\cdot\max\left(1,\frac{2\lambda}{n}\right)}$$

$$\mathrm{Thput}_{\mathrm{br}}(\text{Alg. A.1})(n,\lambda)=\frac{1}{\max(1,\lambda)}$$

Algorithm A.1 Representative for communication history algorithms (uniform)

```
Initialisation:
   $received_p \leftarrow \emptyset$                                {Messages received by process p}
   $delivered_p \leftarrow \emptyset$                               {Messages delivered by process p}
   $deliverable_p \leftarrow \emptyset$                             {Messages ready to be delivered by process p}
   $LC_p[p_1, \ldots, p_n] \leftarrow \{0, \ldots, 0\}$   { $LC_p[q]$: logical clock of process q as seen by process p}

procedure $TO\text{-}broadcast(m)$                             {To TO-broadcast a message m}
   $LC_p[p] \leftarrow LC_p[p] + 1$
   send $(m, LC_p[p])$ to all

when no message sent for $\Delta_{\mathrm{live}}$ time units
   $LC_p[p] \leftarrow LC_p[p] + 1$
   send $(\perp, LC_p[p])$ to all

when receive $(m, ts(m))$
   $LC_p[p] \leftarrow \max(ts(m), LC_p[p]) + 1$                 {Update logical clock}
   $LC_p[sender(m)] \leftarrow ts(m)$
   $received_p \leftarrow received_p \cup \{m\}$
   $deliverable_p \leftarrow \emptyset$
   for each message $m'$ in $received_p \setminus delivered_p$ do
      if $ts(m') < \min_{q \in \pi(t)} LC_p[q]$ then
         $deliverable_p \leftarrow deliverable_p \cup \{m'\}$
   deliver all messages in $deliverable_p$, according to the total order $\Longrightarrow$ (see Sect. 4.3.1)
   $delivered_p \leftarrow delivered_p \cup deliverable_p$
```

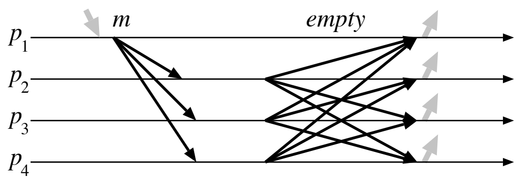


Figure A.1: Uniform communication history

Non-Uniform Privilege-Based

$$
\begin{aligned}
&\mathrm{Latency_{pp}}(\text{Alg. A.2})(n,\lambda)\\
&\quad = \left(\tfrac{n}{2}+1\right)\cdot(2\lambda+1)+(n-2)\cdot\max(1,\lambda)\\
&\quad + \begin{cases}
\max\left(0,n-3\right) & \text{if } \lambda \le \frac{1}{2}\\
\max\left(0,\left\lfloor\frac{n-3}{2}\right\rfloor\right) & \text{if } \frac{1}{2} < \lambda \le 1\\
\left\{\begin{array}{ll}
\begin{array}{l}\max\left(0,2-\lambda+((n-5)\bmod 3)(1-\lambda)\right)\\ \quad+\max\left(0,\left\lfloor\frac{n-5}{3}\right\rfloor(4-3\lambda)\right)\end{array} & \text{if } n>4\\
0 & \text{otherwise}
\end{array}\right\} & \text{if } 1<\lambda\le 2\\
\left\{\begin{array}{ll}
1+\lceil\lambda\rceil-\lambda & \text{if } n>2 \text{ and } (n-2)\bmod\left(2\left\lfloor\lambda\right\rfloor+1\right)=0\\
0 & \text{otherwise}
\end{array}\right\} & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$\mathrm{Latency_{br}}(\text{Alg. A.2})(n,\lambda) = \left(\frac{n}{2}+1\right)\cdot(2\lambda+1)$$

$$\mathrm{Thput_{pp}}(\text{Alg. A.2})(n,\lambda) = \frac{1}{(n-1)\cdot\max\left(1,\frac{2\lambda}{n}\right)}$$

$$\mathrm{Thput_{br}}(\text{Alg. A.2})(n,\lambda) = \frac{1}{\max(1,\lambda)}$$

Algorithm A.2 Representative for privilege-based algorithms (non-uniform).

```
Initialisation:
  $sendQ_p \leftarrow \epsilon$                        {sequence of messages to send (send queue)}
  $recvQ_p \leftarrow \epsilon$                        {sequence of received messages (receive queue)}
  $lastdelivered_p \leftarrow 0$                       {sequence number of the last delivered message}
  $toknext_p \leftarrow p + 1 (\bmod n)$               {identity of the next process along the logical ring}
  if $p = p_1$ then                                    {virtual message to initiate the token rotation}
    send $(\perp, 0, 1)$ to $p_1$                      {format: (message, seq. number, next token holder)}

procedure TO-broadcast$(m)$                            {To TO-broadcast a message $m$}
  $sendQ_p \leftarrow sendQ_p \triangleright m$

when receive $(m, seqnum, tokenholder)$
  if $m \neq \perp$ then                               {Receive new messages}
    $recvQ_p \leftarrow recvQ_p \triangleright (m, seqnum)$
  if $p = tokenholder$ then                            {Circulate token, if appropriate}
    if $sendQ_p \neq \epsilon$ then                    {Send pending messages, if any}
      $msg \leftarrow head.sendQ_p$
      $sendQ_p \leftarrow tail.sendQ_p$
      send $(msg, seqnum + 1, toknext_p)$ to all
      $recvQ_p \leftarrow recvQ_p \triangleright (msg, seqnum + 1)$
    else
      send $(\perp, seqnum, toknext_p)$ to $toknext_p$
  while $\exists m'$ s.t. $(m', lastdelivered_p + 1) \in recvQ_p$ do   {Deliver messages that can be}
    $recvQ_p \leftarrow recvQ_p - \{m'\}$
    deliver $(m')$
    increment$(lastdelivered_p)$
```

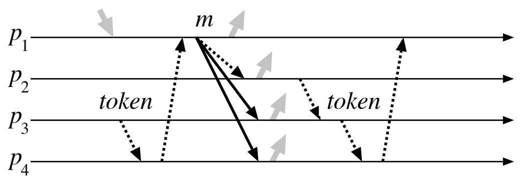


Figure A.2: Non-uniform privilege-based

Uniform Privilege-Based

$$\text{Latency}_{\text{pp}}(\text{Alg. A.3})(n, \lambda) = \frac{5n}{2}(2\lambda + 1) + \begin{cases} (n-2)(1-2\lambda) & \text{if } \lambda \leq \frac{1}{2} \\ \left\lfloor \frac{n-2}{2} \right\rfloor (2-2\lambda) & \text{if } \frac{1}{2} < \lambda \leq 1 \\ (3-2\lambda) \max\left(0, \left\lfloor \frac{n-4}{3} \right\rfloor\right) & \text{if } 1 < \lambda \leq \frac{3}{2} \\ (\lambda - \lfloor \lambda \rfloor) \max\left(0, \left\lfloor \frac{n-4}{3} \right\rfloor\right) & \text{if } \frac{3}{2} < \lambda \leq 2 \\ (\lambda - \lfloor \lambda \rfloor) \max\left(0, \left\lfloor \frac{n-3}{5} \right\rfloor\right) & \text{otherwise} \end{cases}$$

$$\text{Latency}_{\text{br}}(\text{Alg. A.3})(n, \lambda) = \left(\frac{5n}{2} - 1\right) \cdot (2\lambda + 1)$$

$$\text{Thput}_{\text{pp}}(\text{Alg. A.3})(n, \lambda) = \frac{1}{(n-1)\max\left(1, \frac{2\lambda}{n}\right)}$$

$$\text{Thput}_{\text{br}}(\text{Alg. A.3})(n, \lambda) = \frac{1}{\max(1, \lambda)}$$

Algorithm A.3 Representative for privilege-based algorithms (uniform).

```
Initialisation:
  $sendQ_p \leftarrow \epsilon$                              {sequence of messages to send (send queue)}
  $recvQ_p \leftarrow \epsilon$                              {sequence of received messages (receive queue)}
  $stableQ_p \leftarrow \epsilon$                            {sequence of stable messages (stable queue)}
  $lastdelivered_p \leftarrow 0$                             {sequence number of the last delivered message}
  $toknext_p \leftarrow p + 1 (\mathrm{mod}\, n)$            {identity of the next process along the logical ring}
  $acks_p \leftarrow (\emptyset, \ldots, \emptyset)$         {array $[p_1, \ldots, p_n]$ of message sets (acknowledged messages)}
  if $p = p_1$ then                                          {send a virtual message to initiate the token rotation}
    send $(\bot, 0, 1, acks_p)$ to $p_1$                     {format: (message, seq. number, next token holder, acks)}

procedure $TO\text{-}broadcast(m)$                           {To TO-broadcast a message $m$}
  $sendQ_p \leftarrow sendQ_p \rhd m$

when receive $(m, seqnum, tokenholder, acks)$
  if $m \neq \bot$ then                                      {Receive new messages}
    $recvQ_p \leftarrow recvQ_p \rhd (m, seqnum)$
  while $\exists m'$ s.t. $(m', seq') \in recvQ_p$ do        {Ack recv'd messages and detect stability}
    $acks[p] \leftarrow acks[p] \cup \{m'\}$
    if $\forall q \in \pi(t) : m' \in acks[q]$ then
      $stableQ_p \leftarrow stableQ_p \rhd (m', seq')$
      $recvQ_p \leftarrow recvQ_p - \{(m', seq')\}$
  if $p = tokenholder$ then                                  {Circulate token, if appropriate}
    if $sendQ_p \neq \epsilon$ then                          {Send pending messages, if any}
      $msg \leftarrow head.sendQ_p$
      $sendQ_p \leftarrow tail.sendQ_p$
      send $(msg, seqnum + 1, toknext_p, acks)$ to all
      $recvQ_p \leftarrow recvQ_p \rhd (msg, seqnum + 1)$
    else
      send $(\bot, seqnum, toknext_p, acks)$ to $toknext_p$
  while $\exists m'$ s.t. $(m', lastdelivered_p + 1) \in stableQ_p$ do   {Deliver messages that can be}
    $stableQ_p \leftarrow stableQ_p - \{m'\}$
    deliver $(m')$
    increment$(lastdelivered_p)$
```

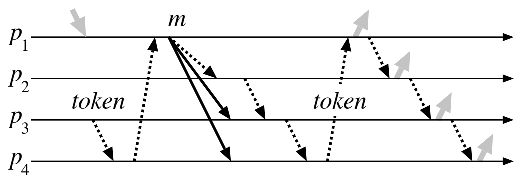


Figure A.3: Uniform privilege-based

Non-Uniform Moving Sequencer

$$\begin{aligned}
&\mathrm{Latency_{pp}}(\text{Alg. A.4})(n,\lambda)\\
&= 2\lambda + 2 + \begin{cases}
\left\{\begin{array}{ll} \lambda+1 & \text{if } n=2 \\ 2n-4 & \text{otherwise} \end{array}\right\} & \text{if } \lambda < \frac{1}{2} \\
\left\{\begin{array}{ll} 2\lambda + (n-2)\max(1,\lambda) & \text{if } n < 4 \\ 2\max(1,\lambda) + \left\{\begin{array}{ll} 2n-6 & \text{if } n > \frac{6-2\lambda}{2-\lambda} \\ (n-2)\lambda & \text{otherwise} \end{array}\right\} & \text{otherwise} \end{array}\right\} & \text{if } \frac{1}{2} \leq \lambda < 2 \\
n\lambda & \text{otherwise}
\end{cases}
\end{aligned}$$

$$\mathrm{Latency_{br}}(\text{Alg. A.4})(n,\lambda) = 4\lambda + 2$$

$$\mathrm{Thput_{pp}}(\text{Alg. A.4})(n,\lambda) = \frac{1}{\max\left(\frac{4n^2-4n-1}{n^2}\lambda, 2n-2\right)}$$

$$\mathrm{Thput_{br}}(\text{Alg. A.4})(n,\lambda) = \frac{1}{2\max(1,\lambda)}$$

Algorithm A.4 Representative for moving sequencer algorithms (non-uniform).

```
Initialisation:
  $recvQ_p \leftarrow \epsilon$                                        {sequence of received messages (receive queue)}
  $seqQ_p \leftarrow \epsilon$                                         {sequence of messages with a seq. number}
  $lastdelivered_p \leftarrow 0$                                       {sequence number of the last delivered message}
  $toknext_p \leftarrow p + 1 (\bmod n)$                               {identity of the next process along the logical ring}
  **if** $p = p_1$ **then**                                            {virtual message to initiate the token rotation}
    send $(\perp, 0, 1)$ to $p_1$                                      {format: (message, seq. number, next token holder)}

**procedure** $TO\text{-}broadcast(m)$                                  {To TO-broadcast a message $m$}
  send $(m)$ to all
  $recvQ_p \leftarrow recvQ_p \rhd m$

**when** receive $(m)$
  $recvQ_p \leftarrow recvQ_p \rhd m$

**when** receive $(m, seqnum, tokenholder)$
  **if** $m \neq \perp$ **then**                                       {Receive new sequence number}
    $seqQ_p \leftarrow seqQ_p \rhd (m, seqnum)$
  **if** $p = tokenholder$ **then**                                    {Circulate token, if appropriate}
    **wait until** $(recvQ_p \setminus seqQ_p) \neq \epsilon$
    $msg \leftarrow$ select first $msg$ in $recvQ_p$ s.t. $(msg, -) \notin seqQ_p$
    send $(msg, seqnum + 1, toknext_p)$ to all
    $seqQ_p \leftarrow seqQ_p \rhd (msg, seqnum + 1)$
  **while** $\exists m'$ s.t. $(m', lastdelivered_p + 1) \in seqQ_p \wedge m' \in recvQ_p$ **do**   {Deliver messages that can be}
    $seqQ_p \leftarrow seqQ_p - \{(m', -)\}$
    $recvQ_p \leftarrow recvQ_p - \{m'\}$
    deliver $(m')$
    increment$(lastdelivered_p)$
```

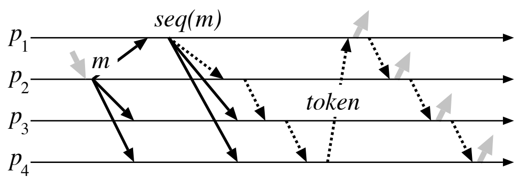


Figure A.4: Non-uniform moving sequencer

Uniform Moving Sequencer

$$
\begin{aligned}
&\text{Latency}_{\text{pp}}(\text{Alg. A.5})(n,\lambda)\\
&\quad = 2(n-1)(2\lambda+1)\\
&\quad + \left\{\begin{array}{ll}
\left\{\begin{array}{ll}
2(n-\lambda-1) & \text{if } n > 4\\
\left\{\begin{array}{ll}
2\left(\max(n-2, 2\lambda)+1\right) & \text{if } \lambda \geq \frac{1}{2}\\
\left\{\begin{array}{ll}
6-2\lambda & \text{if } n = 4\\
4 & \text{if } n = 3\\
2+4\lambda & \text{if } n = 2
\end{array}\right\} & \text{otherwise}
\end{array}\right\} & \text{otherwise}
\end{array}\right\} & \text{if } \lambda < 1\\
\\
\left\{\begin{array}{ll}
\begin{array}{l}
2\lambda + \max\left((n-2)\lambda, 2\lambda+2\right)\\
\qquad + \left\{\begin{array}{ll}
2n-6-(n-2)\lambda & \text{if } n \geq \frac{6-2\lambda}{2-\lambda}\\
2-\lambda & \text{if } n < \frac{5-2\lambda}{2-\lambda}\\
2n-5-(n-2)\lambda & \text{otherwise}
\end{array}\right\}
\end{array} & \text{if } n > 5\\
2\lambda+1+\max(2\lambda+1, 2+\lceil\lambda\rceil) & \text{if } n = 5\\
4\lambda+2 & \text{otherwise}
\end{array}\right\} & \text{if } 1 \leq \lambda < 2\\
\\
2\lambda + \max\left((n-2)\lambda, 2\lambda+2\right) & \text{otherwise}
\end{array}\right.
\end{aligned}
$$

$$\text{Latency}_{\text{br}}(\text{Alg. A.5})(n,\lambda) = 4n\lambda + 2n$$

$$\text{Thput}_{\text{pp}}(\text{Alg. A.5})(n,\lambda) = \frac{1}{\max\left(\frac{4n^2-4n-1}{n^2}\lambda, 2n-2\right)}$$

$$\text{Thput}_{\text{br}}(\text{Alg. A.5})(n,\lambda) = \frac{1}{2\max(1,\lambda)}$$

Algorithm A.5 Representative for moving sequencer algorithms (uniform).

```
Initialisation:
  $recvQ_p \leftarrow \epsilon$                    {sequence of received messages (receive queue)}
  $seqQ_p \leftarrow \epsilon$                     {sequence of messages with a seq. number}
  $stableQ_p \leftarrow \epsilon$                  {sequence of stable messages (stable queue)}
  $lastdelivered_p \leftarrow 0$                   {sequence number of the last delivered message}
  $toknext_p \leftarrow p + 1 (\mathrm{mod} n)$    {identity of the next process along the logical ring}
  $acks_p \leftarrow (\emptyset, \ldots, \emptyset)$   {array $[p_1, \ldots, p_n]$ of message sets (acknowledged messages)}
  **if** $p = p_1$ **then**                        {virtual message to initiate the token rotation}
    send $(\bot, 0, 1, acks_p)$ to $p_1$           {format: (message, seq. number, next token holder, acks)}

**procedure** $TO\text{-}broadcast(m)$             {To TO-broadcast a message $m$}
  send $(m)$ to all
  $recvQ_p \leftarrow recvQ_p \rhd m$

**when** receive $(m)$
  $recvQ_p \leftarrow recvQ_p \rhd m$

**when** receive $(m, seqnum, tokenholder, acks)$
  $acks_p \leftarrow acks_p \cup acks$
  **if** $m \neq \bot$ **and** $m \in recvQ_p$ **then**
    $seqQ_p \leftarrow seqQ_p \rhd (m, seqnum)$
  **while** $\exists (m', seq') \in seqQ_p$ s.t. $m' \in recvQ_p$ **do**   {Ack recv'd messages and detect stability}
    $acks_p[p] \leftarrow acks_p[p] \cup \{m'\}$
    **if** $\forall q \in \pi(t) : m' \in acks_p[q]$ **then**
      $stableQ_p \leftarrow stableQ_p \rhd (m', seq')$
      $seqQ_p \leftarrow seqQ_p - \{(m', -)\}$
      $recvQ_p \leftarrow recvQ_p - \{m'\}$
  **if** $p = tokenholder$ **then**                {Circulate token, if appropriate}
    **wait until** $recvQ_p \neq \epsilon \wedge seqQ_p \neq \epsilon \wedge stableQ_p \neq \epsilon$
    **if** $(recvQ_p \setminus seqQ_p) = \epsilon$ **then**
      send $(\bot, seqnum, toknext_p, acks_p)$ to $toknext_p$
    **else**
      $msg \leftarrow$ select first message $msg$ such that $msg \in recvQ_p \wedge (msg, -) \notin seqQ_p$
      send $(msg, seqnum + 1, toknext_p, acks_p)$ to all
  **while** $\exists m'$ s.t. $(m', lastdelivered_p + 1) \in stableQ_p$ **do**   {Deliver messages that can be}
    $stableQ_p \leftarrow stableQ_p - \{(m', -)\}$
    deliver $(m')$
    increment$(lastdelivered_p)$
```

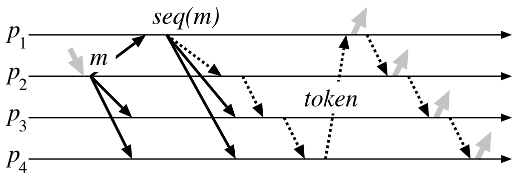


Figure A.5: Uniform moving sequencer

Non-Uniform Fixed Sequencer

$$\mathrm{Latency}_{\mathrm{pp}}(\text{Alg. A.6})(n, \lambda) = 2(2\lambda + 1) + (n - 2)\max(1, \lambda)$$

$$\mathrm{Latency}_{\mathrm{br}}(\text{Alg. A.6})(n, \lambda) = 4\lambda + 2$$

$$\mathrm{Thput}_{\mathrm{pp}}(\text{Alg. A.6})(n, \lambda) = \frac{n}{(n^2 - 1)\max(1, \lambda)}$$

$$\mathrm{Thput}_{\mathrm{br}}(\text{Alg. A.6})(n, \lambda) = \frac{n}{(2n - 1)\max(1, \lambda)}$$

Algorithm A.6 Representative for fixed sequencer algorithms (non-uniform)

```
Code of sequencer:
  Initialisation:
    $seqnum \leftarrow 0$                                    {last seq. number attributed to a message}

  procedure $TO\text{-}broadcast(m)$                        {To TO-broadcast a message $m$}
    $\text{increment}(seqnum)$
    send $(m, seqnum)$ to all
    deliver $(m)$

  when receive $(m)$
    $TO\text{-}broadcast(m)$

Code of all processes except sequencer:
  Initialisation:
    $lastdelivered_p \leftarrow 0$                           {sequence number of the last delivered message}
    $received_p \leftarrow \emptyset$                         {set of received yet undelivered messages}

  procedure $TO\text{-}broadcast(m)$                        {To TO-broadcast a message $m$}
    send $(m)$ to sequencer

  when receive $(m, seq(m))$
    $received_p \leftarrow received_p \cup \{(m, seq(m))\}$
    while $\exists m', seq$ s.t. $(m', seq) \in received_p \wedge seq = lastdelivered_p + 1$ do
      deliver $(m')$
      $\text{increment}(lastdelivered_p)$
      $received_p \leftarrow received_p \setminus \{(m', seq)\}$
```

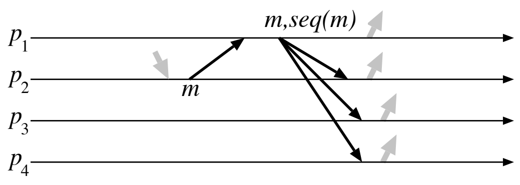


Figure A.6: Non-uniform fixed sequencer

Uniform Fixed Sequencer

$$
\begin{aligned}
&\mathrm{Latency_{pp}}(\text{Alg. A.7})(n,\lambda)\\
&\quad = 2\lambda + 1\\
&\quad + \begin{cases}
3(n-1) + 4\lambda + 2\lambda\,((n-1) \bmod 2) & \text{if } \lambda < \frac{1}{2}\\[2ex]
3(n-1) + 4\lambda + \left\{\begin{array}{ll} 2\lambda & \text{if } n \bmod 3 = 2\\ 2\lambda - 1 & \text{if } n \bmod 3 = 0\\ 0 & \text{otherwise}\end{array}\right\} & \text{if } \frac{1}{2} \le \lambda < 1\\[2ex]
\begin{array}{l}(3n-2)\lambda + 1\\ \quad + \left\{\begin{array}{ll} 2 + (4-n)\lambda & \text{if } n < 5\\ 4 - 2\lambda & \text{if } n = 5\\ \max\left(0, \begin{array}{l}\left\{\begin{array}{ll} \lambda + 1 & \text{if } n \bmod 4 = 0\\ 2 & \text{if } n \bmod 4 = 1\\ 3 & \text{otherwise}\end{array}\right\}\\ -(n-4)(2\lambda - 2)\end{array}\right) & \text{otherwise}\end{array}\right\}\end{array} & \text{if } 1 \le \lambda < \frac{3}{2}\\[2ex]
(3n-2)\lambda + 1 + \left\{\begin{array}{ll} 2 + (4-n)\lambda & \text{if } n < 5\\ \max(0, 4 - 2\lambda) & \text{if } n = 5\\ 0 & \text{otherwise}\end{array}\right\} & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$\mathrm{Latency_{br}}(\text{Alg. A.7})(n,\lambda) = 8\lambda + 4 + (n-2)\max(1,\lambda)$$

$$\mathrm{Thput_{pp}}(\text{Alg. A.7})(n,\lambda) = \frac{n}{(3n^2 - 2n - 1)\max(1,\lambda)}$$

$$\mathrm{Thput_{br}}(\text{Alg. A.7})(n,\lambda) = \frac{n}{(n^2 + 2n - 1)\max(1,\lambda)}$$

Algorithm A.7 Representative for fixed sequencer algorithms (uniform)

```
Code of sequencer:
  Initialisation:
    $seqnum \leftarrow 0$                          {last seq. number attributed to a message}
    $lastdelivered \leftarrow 0$                   {sequence number of the last delivered message}
    $received \leftarrow \emptyset$                {set of received messages}
    $stable \leftarrow \emptyset$                  {set of stable yet undelivered messages}

  procedure $TO\text{-}broadcast(m)$               {To TO-broadcast a message $m$}
    increment($seqnum$)
    send $(m, seqnum)$ to all
    $received \leftarrow received \cup \{(m, seqnum)\}$
    spawn
      wait until $\forall q \in \pi(t)$ : received $(m, seq, ack)$
      send $(m, seq, stable)$ to all
      $stable \leftarrow stable \cup \{(m, seq(m))\}$
      while $\exists (m', seq') \in stable$ s.t. $seq' = lastdelivered + 1$ do
        deliver $(m')$
        increment($lastdelivered$)
        $stable \leftarrow stable \setminus \{(m', seq')\}$

  when receive $(m)$
    $TO\text{-}broadcast(m)$

Code of all processes except sequencer:
  Initialisation:
    $lastdelivered_p \leftarrow 0$                 {sequence number of the last delivered message}
    $received_p \leftarrow \emptyset$              {set of received messages}
    $stable_p \leftarrow \emptyset$                {set of stable yet undelivered messages}

  procedure $TO\text{-}broadcast(m)$               {To TO-broadcast a message $m$}
    send $(m)$ to sequencer

  when receive $(m, seq)$
    $received_p \leftarrow received_p \cup \{(m, seq)\}$
    send $(m, seq, ack)$ to sequencer

  when receive $(m, seq, stable)$
    $stable_p \leftarrow stable_p \cup \{(m, seq)\}$
    while $\exists (m', seq') \in stable_p$ s.t. $seq' = lastdelivered_p + 1$ do
      deliver $(m')$
      increment($lastdelivered_p$)
      $stable_p \leftarrow stable_p \setminus \{(m', seq')\}$
```

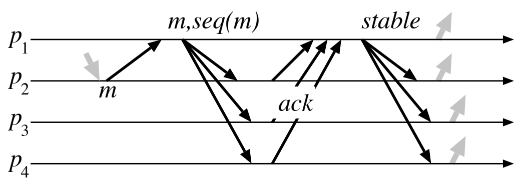


Figure A.7: Uniform fixed sequencer

Non-Uniform Destinations Agreement

$$\mathrm{Latency_{pp}}(\text{Alg. A.8})(n,\lambda)$$
$$= \begin{cases} 3(n-1)+4\lambda+2\lambda\left((n-1) \bmod 2\right) & \text{if } \lambda < \frac{1}{2} \\ 3(n-1)+4\lambda+\left\{\begin{array}{ll} 2\lambda & \text{if } n \bmod 3 = 2 \\ 2\lambda-1 & \text{if } n \bmod 3 = 0 \\ 0 & \text{otherwise} \end{array}\right\} & \text{if } \frac{1}{2} \le \lambda < 1 \\ \begin{array}{l} (3n-2)\lambda+1 \\ \quad + \left\{\begin{array}{ll} 2+(4-n)\lambda & \text{if } n<5 \\ 4-2\lambda & \text{if } n=5 \\ \max\left(0, \begin{array}{l} \left\{\begin{array}{ll} \lambda+1 & \text{if } n \bmod 4 = 0 \\ 2 & \text{if } n \bmod 4 = 1 \\ 3 & \text{otherwise} \end{array}\right\} \\ -(n-4)(2\lambda-2) \end{array}\right) & \text{otherwise} \end{array}\right\} \end{array} & \text{if } 1 \le \lambda < \frac{3}{2} \\ (3n-2)\lambda+1+\left\{\begin{array}{ll} 2+(4-n)\lambda & \text{if } n<5 \\ \max(0,4-2\lambda) & \text{if } n=5 \\ 0 & \text{otherwise} \end{array}\right\} & \text{otherwise} \end{cases}$$

$$\mathrm{Latency_{br}}(\text{Alg. A.8})(n,\lambda) = 6\lambda+3+(n-2)\max(1,\lambda)$$

$$\mathrm{Thput_{pp}}(\text{Alg. A.8})(n,\lambda) = \frac{1}{(3n-3)\max\left(1,\frac{2\lambda}{n}\right)}$$

$$\mathrm{Thput_{br}}(\text{Alg. A.8})(n,\lambda) = \frac{1}{\max\left(n+1,\frac{4n-2}{n}\lambda\right)}$$

Algorithm A.8 Representative for destinations agreement algorithms (non-uniform)

```
Initialisation:
  $received_p \leftarrow \emptyset$        {set of messages received by process p, with a temporary local timestamp}
  $stamped_p \leftarrow \emptyset$         {set of messages received by process p, with a final global timestamp}
  $LC_p \leftarrow 0$                      {$LC_p$: logical clock of process p}

procedure $TO\text{-}broadcast(m)$         {To TO-broadcast a message m}
  send $(m, nodeliver)$ to all (including itself)
  spawn
    wait until $\forall q \in \pi(t)$ : received $(m, ts_q(m))$
    $TS(m) \leftarrow \max_{q \in \pi(t)} ts_q(m)$
    send $(m, TS(m), deliver)$ to all (including itself)

when receive $(m, nodeliver)$
  $ts_p(m) \leftarrow LC_p$
  $received_p \leftarrow received_p \cup \{(m, ts_p(m))\}$
  send $(m, ts_p(m))$ to $sender(m)$
  $LC_p \leftarrow LC_p + 1$

when receive $(m, TS(m), deliver)$
  $stamped_p \leftarrow stamped_p \cup \{(m, TS(m))\}$
  $received_p \leftarrow received_p \setminus \{(m, -)\}$
  $deliverable \leftarrow \emptyset$
  for each $m'$ in $stamped_p$ such that $\forall m'' \in received_p : TS(m') < ts_p(m'')$ do
    $deliverable \leftarrow deliverable \cup \{(m', TS(m'))\}$
  deliver all messages in $deliverable$ in increasing order of $TS(m)$
  $stamped_p \leftarrow stamped_p \setminus deliverable$
```

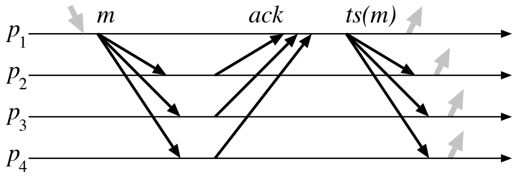
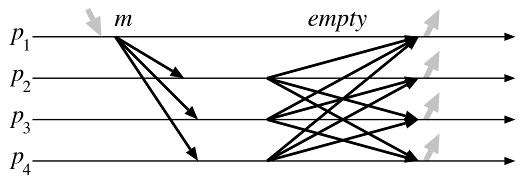
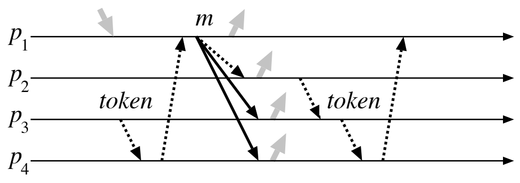


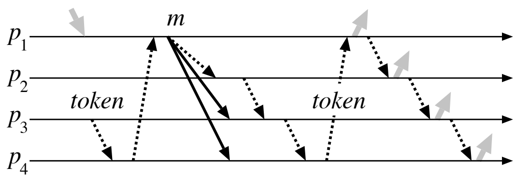
Figure A.8: Non-uniform destinations agreement



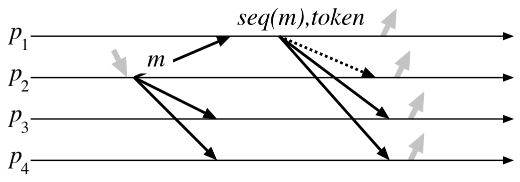
(a) Communication history (uniform)



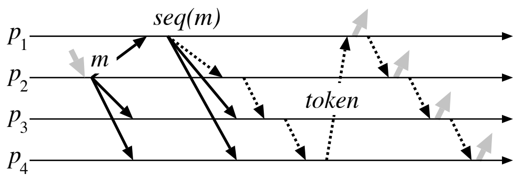
(b) privilege-based (non-uniform)



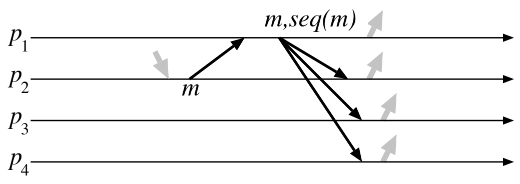
(c) privilege-based (uniform)



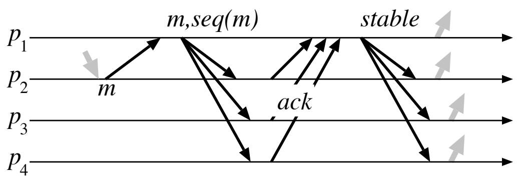
(d) Moving sequencer (non-uniform)



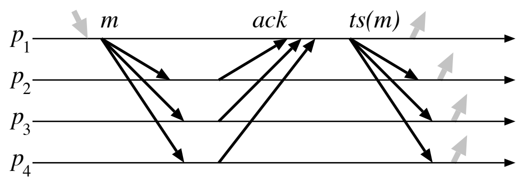
(e) Moving sequencer (uniform)



(f) Fixed sequencer (non-uniform)



(g) Fixed sequencer (uniform)



(h) Destinations agreement (non-uniform)

Figure A.9: Graphical representation of representative algorithms (summary)

Curriculum Vitæ

Xavier Défago was born in Basel (Switzerland) in 1970. He attended primary and secondary school at several places: Basel (CH), Gex (F), and Vevey (CH). He obtained his *Baccalaureat* and *Maturité fédérale* in 1989 from the CESSEV high school in La Tour-de-Peilz (CH). From 1989 to 1995, he studied computer science at the Swiss Federal Institute of Technology in Lausanne (EPFL). During this period, he also worked several times for the Swiss branch of Nestlé SA in Vevey. In 1995 and 1996, he worked for a year at the C&C Central Research Labs of NEC Corporation in Kawazaki (Japan) under the supervision of Dr Akihiko Konagaya, where he developed a prototype of a Java virtual machine for massively parallel computers. Since 1996, he has been working in the Operating Systems Lab at the EPFL as a research and teaching assistant, and as a Ph.D. student, under the guidance of Professor André Schiper.

Refereed Publications

[1] P. Urbán, X. Défago, and A. Schiper. **Contention-aware metrics for distributed algorithms: comparison of Atomic Broadcast algorithms.** In *Proc. of the 9th IEEE Int'l Conf. on Computer Communications and Networks (ICCCN'2000)*, Las Vegas, NE, USA, October 2000.

[2] P. Felber, X. Défago, R. Guerraoui, and P. Oser. **Failure detectors as first class objects.** In *Proc. of the 1st OMG Int'l Symp. on Distributed Objects and Applications (DOA)*, Edinburgh, Scotland, September 1999.

[3] P. Felber, X. Défago, P. Eugster, and A. Schiper. **Replicating CORBA objects: a marriage between active and passive replication.** In *Proc. of the 2nd IFIP Int'l Working Conf. on Distributed Applications and Interoperable Systems (DAIS-2)*, Helsinki, Finland, June 1999.

[4] X. Défago, P. Felber, and A. Schiper. **Optimization techniques for replicating CORBA objects.** In *Proc. of the 4th IEEE Int'l Workshop on Object-oriented Real-time Dependable Systems (WORDS-4)*, Santa Barbara, CA, January 1999.

[5] X. Défago, A. Schiper, and N. Sergent. **Semi-passive replication.** In *Proc. of the 17th IEEE Int'l Symp. on Reliable Distributed Systems (SRDS-17)*, West Lafayette, IN, October 1998.

[6] X. Défago, K. R. Mazouni, and A. Schiper. **Highly available trading system: Experiments with CORBA.** In *Middleware'98: 1st IFIP Int'l Conf. on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, UK, September 1998.

[7] X. Défago, P. Felber, B. Garbinato, and R. Guerraoui. **Reliability with CORBA event channels.** In *Proc. of the 3rd USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, Portland, OR, June 1997.

[8] U. G. Wilhelm and X. Défago. **Objets protégés cryptographiquement.** (*cryptographically protected objects*, in French). In *Actes RenPar'9*, Lausanne, Switzerland, May 1997.

[9] X. Défago and A. Konagaya. **Issues in building a parallel Java virtual machine on Cenju-3/DE.** In *Proc. of the 1st Cenju Workshop, High Performance Computing Asia (HPC'Asia)*, Seoul, Korea, April 1997.

[10] X. Défago. **Reliability issues with CORBA event channels.** In *Proc. of the 2nd European Research Seminar on Advances in Distributed Systems (ERSADS-2)*, Zinal, Switzerland, March 1997.

[11] B. Garbinato, X. Défago, R. Guerraoui, and K. R. Mazouni. **Abstractions pour la programmation concurrente dans GARF.** (*Abstractions for concurrent programming in GARF*, in French). *Calculateurs parallèles*, 6(2):85–98, June 1994.

Technical Reports

[12] X. Défago and A. Schiper. **Semi-passive replication and Lazy Consensus.** TR DSC/2000/027, Communication Systems Department, Swiss Federal Institute of Technology, Lausanne, Switzerland, May 2000.

[13] P. Urbán, X. Défago, and A. Schiper. **Contention-aware metrics: analysis of distributed algorithms.** TR DSC/2000/012, Communication Systems Department, Swiss Federal Institute of Technology, Lausanne, Switzerland, February 2000.

[14] N. Sergent, X. Défago, and A. Schiper. **Failure detectors: implementation issues and impact on consensus performance.** TR SSC/1999/019, Communication Systems Division, Swiss Federal Institute of Technology, Lausanne, Switzerland, May 1999.

[15] X. Défago, A. Schiper, and N. Sergent. **Semi-passive replication.** TR 98/277, Computer Science Department, Swiss Federal Institute of Technology, Lausanne, Switzerland, May 1998.

[16] X. Défago, P. Felber, and R. Guerraoui. **Reliable CORBA event channels.** TR 97/229, Computer Science Department, Swiss Federal Institute of Technology, Lausanne, Switzerland, May 1997.

[17] X. Défago and A. Konagaya. **A parallel Java virtual machine.** In *Information Processing Society of Japan, 53rd National Meeting*, Osaka, Japan, September 1996.

[18] X. Défago. **Towards a parallel Java virtual machine.** TR LR-7212, Computer System Research Lab., C&C Research Labs, NEC Corp., Kawazaki, Japan, July 1996.