

ATOMIC BROADCAST: A FAULT-TOLERANT TOKEN BASED ALGORITHM AND PERFORMANCE EVALUATIONS

THÈSE N° 3811 (2007)

PRÉSENTÉE LE 31 MAI 2007

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES RÉPARTIS
SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Nils Richard EKWALL

ingénieur informaticien diplômé EPF
de nationalités suédoise et suisse et originaire de Rosières (SO)

acceptée sur proposition du jury:

Prof. K. Aberer, président du jury
Prof. A. Schiper, directeur de thèse
Prof. D. Malkhi, rapporteur
Prof. N. Suri, rapporteur
Prof. W. Zwaenepoel, rapporteur



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL

2007

Abstract

Within only a couple of generations, the so-called digital revolution has taken the world by storm: today, almost all human beings interact, directly or indirectly, at some point in their life, with a computer system. Computers are present on our desks, computer systems control the antilock braking system and the stability control in cars, they collect usage statistics in elevators in order to anticipate maintenance and repair operations. Computer systems also operate critical systems, such as nuclear power plants, airplane control systems or space rockets. Furthermore, computer systems are not only omnipresent, but also increasingly networked.

As the use of computer systems has increased dramatically over the past decades, the needs and expectations associated with these systems have also increased. In particular, one of the critical points of a system is its availability (the fraction of the time during which the system provides a service to the users): the costs and negative publicity of a system outage (be it a commercial web site or a stock exchange for example) are often considerable.

Fault tolerance is one of the approaches to designing a highly-available system: a fault tolerant system is designed in such a way that the failure of one of the components of the system does not compromise the functionality of the system as a whole. Replication is one of the common fault tolerance techniques. Instead of having a single machine (a *replica*) providing a service, the system is composed of several replicas running the service and connected through a network. If one of the replicas fails, the service is still provided by the remaining replicas. The replication technique is interesting as it can be achieved by using software running on commodity hardware, thus avoiding the high cost of special purpose hardware.

Replication, although intuitive to understand, is complex to implement in practice, as the replicas have to interact in order to ensure the consistency of the system as a whole. Group communication simplifies the replication problem, by hiding issues such as the communication between the replicas, the crashes of one or several replicas and the synchronization of the replicas.

In this thesis, we start by comparing two replication techniques — group communication and quorum systems — and identifying in which case either technique should be used. Atomic broadcast (a group communication primitive at the heart of this work) allows replicas to broadcast messages

to each other and then deliver them in the same total order, even if replicas broadcast messages quasi simultaneously. Atomic broadcast is especially useful for replication: since all replicas deliver messages in the same order, their state is kept consistent.

After the comparison between the replication techniques, we present an atomic broadcast algorithm designed to perform well when the system is heavily loaded and that allows to quickly detect crashed replicas (by minimizing the consequences of wrongly suspecting a non-crashed replica). The presentation of the algorithm includes simulation results comparing the performance of the new algorithm to previously proposed atomic broadcast algorithms.

The second part of the thesis focuses on the experimental performance evaluation of the new algorithm in several settings. We start by comparing four atomic broadcast algorithms in a local area network. We then compare three of the four algorithms in a wide area network, with sites in Switzerland, Japan and France, and where the round trip time between the sites varies between 4 and 300 *ms*. Finally, we evaluate the impact of the size of the system (the number of replicas) on the performance of the algorithms.

Keywords: replication, fault tolerance, distributed algorithms, group communication, agreement problems, consensus, atomic broadcast, token based algorithms, failure detectors, performance evaluation, scalability, local area network, wide area network.

Résumé

En seulement deux générations, la révolution numérique a conquis le monde : aujourd'hui, quasiment tous les êtres humains interagissent, directement ou indirectement, à un moment de leur vie, avec un système informatique. Les ordinateurs sont présents sur nos bureaux ; les systèmes informatiques gèrent les freins ABS et le contrôle de la trajectoire de nos voitures et collectent des statistiques d'utilisation dans les ascenseurs afin d'anticiper les opérations de maintenance et de réparation. L'informatique est également utilisée dans les systèmes critiques, tels que les centrales nucléaires, le contrôle aérien ou les fusées spatiales. De plus, les systèmes informatiques sont non seulement omniprésents, mais de plus en plus souvent reliés en réseau.

Avec l'augmentation considérable de l'utilisation de systèmes informatiques, les besoins et attentes associés à ces systèmes ont aussi augmenté. En particulier, un des points critiques d'un système est sa disponibilité (la fraction du temps durant laquelle le système fournit un service aux utilisateurs) : les coûts et la publicité négative d'une panne du système (que ce soit un serveur web commercial ou un marché financier) sont souvent considérables.

La tolérance aux pannes est une des approches pour concevoir un système à haute disponibilité : un système tolérant aux pannes est conçu de telle manière à ce que la panne d'un des composants du système ne compromette pas la fonctionnalité du système dans son ensemble. La répliction est une des techniques communes de tolérance aux pannes. Au lieu d'avoir une seule machine (une *réplique*) qui fournit un service, le système est composé de plusieurs répliques, reliées à un réseau et qui fournissent chacune le service. Si une des répliques tombe en panne, le service est toujours assuré par les répliques restantes. La technique de la réplication est intéressante, car elle peut être mise en œuvre avec des logiciels qui s'exécutent sur du matériel informatique standard, évitant ainsi le coût élevé associé au matériel informatique conçu sur mesure.

La réplication, bien qu'intuitive, est complexe à implémenter en pratique, étant donné que les répliques doivent interagir afin d'assurer la cohérence du système en entier. La communication de groupe simplifie la réplication, en cachant les problèmes tels que la communication entre les répliques, les pannes d'une ou plusieurs répliques et la synchronisation entre répliques.

Cette thèse débute par une comparaison de deux techniques de réplication — la communication de groupe et les systèmes de quorum — qui identifie dans quel cas une technique est mieux adaptée que l'autre. La diffusion atomique (une primitive de communication de groupe au cœur de ce travail) permet aux répliques de diffuser des messages entre eux et de tous les délivrer dans le même ordre, même si certaines répliques diffusent des messages quasi-simultanément. La diffusion atomique est particulièrement utile pour la réplication : puisque toutes les répliques délivrent les messages dans le même ordre, la cohérence de leur état est maintenu.

Après la comparaison des deux techniques de réplication, nous présentons un algorithme de diffusion atomique conçu pour de bonnes performances lors d'une charge élevée sur le système, et qui permette de rapidement détecter les pannes éventuelles des répliques (en minimisant les conséquences négatives d'une suspicion à tort d'une réplique qui n'est pas en panne). La présentation de l'algorithme inclut des résultats de simulations qui comparent la performance du nouvel algorithme par rapport à d'autres algorithmes de diffusion atomique proposés par le passé.

La seconde partie de cette thèse se focalise sur l'évaluation expérimentale de la performance du nouvel algorithme dans plusieurs situations. Nous commençons par comparer quatre algorithmes de diffusion atomique dans un réseau local. Nous comparons ensuite trois des quatre algorithmes dans un réseau étendu, avec des sites en Suisse, au Japon et en France, et où les temps aller retour entre sites varient de 4 à 300 *ms*. Finalement, nous évaluons l'impact de la taille du système (le nombre de répliques) sur la performance des algorithmes.

Mots-clés : réplication, tolérance aux pannes, algorithmes répartis, communication de groupe, problèmes d'accord, consensus, diffusion atomique, algorithmes basés sur des jetons, détecteurs de panne, évaluation de performance, extensibilité, réseau local, réseau étendu.

Acknowledgments

A thesis is usually a long endeavor that requires countless hours of work and support to be completed. This thesis was no exception to this rule, and there are thus many people without whom this work would not have been possible.

First and foremost, I would like to thank my supervisor, Prof. André Schiper, for his trust and advice throughout these five years at the Distributed Systems Laboratory (LSR). André gave me the opportunity to participate in the REMUNE IST European project, allowing me to get a glimpse of the processes and inner workings of a large-scale industrial project, while still working in an academic environment. Thesis supervisors with André's qualities are scarce and it was truly a privilege completing this thesis under his guidance.

I would also like to thank Péter Urbán, who supervised my first student project at LSR. What started as a semester project eventually became the Chapter 6 of this thesis. While working with Péter, I also gained an initial insight into the issues of performance evaluation, a theme that is central to this thesis.

Reviewing a thesis requires a significant amount of time and effort. I therefore wish to express my appreciation to the members of the thesis jury: Prof. Dahlia Malkhi, Prof. Neeraj Suri and Prof. Willy Zwaenepoel, as well as the president of the jury, Prof. Karl Aberer.

Furthermore, I am grateful to Prof. Xavier Défago and Matthias Wiesmann, as well as Prof. Pierre Sens, for the access to the infrastructures in Japan and France respectively, which was essential for the performance evaluations in Chapters 8 and 9.

My gratitude also goes to all the colleagues and friends from LSR, with whom I had a chance to work, share a couple of beers at Satellite or enjoy one of the lab's ski days. In particular, I would like to thank David Cavin, Sergio Mena, Yoav Sasson, Arnas Kupšys, Stefan Pleisch, Olivier Rütli, Paweł Wojciechowski, Fatemeh Borran-Dejnabadi, Martin Hutle and Nuno Santos. I am also very grateful to France Faille for her kind help in dealing with all the administrative issues that arose during these years.

Last, but definitely not least, I wish to thank all my friends and family for their support and the good times during these years at EPFL. Without them, this adventure would by far not have been as enjoyable as it was. I especially thank Céline for these wonderful years side by side, my parents for their continuous encouragements and my brother, Thomas, for the great moments together.

Acknowledgments

Contents

1	Introduction	1
1.1	Research context and motivation	1
1.2	Research contributions	4
1.3	Structure of the thesis	6
2	Replication: understanding the advantage of atomic broadcast over quorum systems	9
2.1	Different isolation degrees	10
2.2	No isolation	11
2.3	“Read-write” isolation only	12
2.3.1	Read-write isolation and the consensus problem . . .	12
2.3.2	Implementing read-write isolation with quorums . .	13
2.3.3	Implementing read-write isolation with atomic broadcast	13
2.3.4	Discussion	15
2.4	General isolation	15
2.5	Discussion	16

Part I Solving Atomic Broadcast with Failure Detectors and Token based Algorithms

3	System models and definitions	21
3.1	System models	21
3.1.1	Failure modes	21
3.1.2	Synchrony	23
3.1.3	The Heard-Of model	27
3.2	Agreement problems	28
3.2.1	Consensus	29
3.2.2	Reliable broadcast	29

3.2.3	Atomic broadcast	30
4	Token based atomic broadcast	31
4.1	Token based atomic broadcast using unreliable failure detectors	31
4.1.1	Introduction	31
4.1.2	System model and definitions	34
4.1.3	Failure detector \mathcal{R}	35
4.1.4	Token based consensus	37
4.1.5	Token based atomic broadcast algorithms	43
4.1.6	Simulation Results	47
4.1.7	Related work	58
4.1.8	Discussion	58
4.2	Variants and optimizations of the token based algorithm	59
4.2.1	Unbounded $adeliv_i$ set of ordered messages	60
4.2.2	Bounded-size $adeliv_i$ set of ordered messages	61
4.3	Adapting the algorithm to the Heard-Of model	64
4.3.1	Description of the algorithm	65
4.3.2	Heard-Of predicates	68
4.3.3	Discussion	69
5	Solving atomic broadcast with indirect consensus	71
5.1	Motivation and indirect consensus	72
5.1.1	Atomic broadcast on message identifiers	72
5.1.2	Violating the <i>Validity</i> of atomic broadcast	74
5.1.3	Indirect consensus	75
5.1.4	Reducing atomic broadcast to indirect consensus	76
5.2	Solving indirect consensus	77
5.2.1	Conditions on the correctness of indirect consensus algorithms	77
5.2.2	Adapting Chandra-Toueg’s $\diamond\mathcal{S}$ consensus algorithm	79
5.2.3	Adapting Mostéfaoui-Raynal’s $\diamond\mathcal{S}$ consensus algorithm	83
5.3	Performance measurements	88
5.3.1	System setup and the Neko framework	88
5.3.2	Performance metric: latency versus throughput and message size	88
5.3.3	Performance results: overhead of indirect consensus	90
5.3.4	Performance results: comparison of two correct approaches	91

5.3.5	Overview of the performance results	91
5.4	Discussion	93

Part II Experimental Evaluation of Atomic Broadcast Algorithms

6	Robust TCP connections for fault tolerant computing	97
6.1	Design of the protocol	99
6.1.1	Requirements	99
6.1.2	Issues at the session layer	100
6.1.3	The problem of control messages	101
6.2	The session layer protocol	103
6.2.1	Opening a connection and reconnection	104
6.2.2	Data exchange	104
6.2.3	Closing the connection	105
6.2.4	Handling TCP errors	105
6.2.5	UDP control messages	105
6.3	Java Implementation	106
6.3.1	Classes	106
6.3.2	Integration into Java	107
6.4	Performance	107
6.5	Related work	109
6.6	Discussion	110
7	Comparing atomic broadcast algorithms in a local area network	111
7.1	Algorithms	112
7.1.1	Chandra-Toueg atomic broadcast algorithm	112
7.1.2	Moving sequencer atomic broadcast algorithm	113
7.2	Elements of our performance study	114
7.2.1	Performance metrics and workloads	115
7.2.2	Faultloads	115
7.2.3	Implementation framework and issues	117
7.2.4	Evaluation environment	118
7.3	Results	118
7.3.1	Comparing failure detector based implementations	118
7.3.2	Comparing token based implementations	125

7.4	Discussion	130
-----	----------------------	-----

8 Modeling and validating the performance of atomic broadcast algorithms in high latency networks 133

8.1	Motivation and Related Work	134
8.1.1	The trade-off between number of messages and communication steps	135
8.1.2	Related work	136
8.2	System model	137
8.2.1	Reliable broadcast, consensus and atomic broadcast	137
8.2.2	Two consensus algorithms	138
8.2.3	Two atomic broadcast algorithms	138
8.3	Performance metrics and workloads	140
8.3.1	Performance metric – latency vs. throughput:	140
8.3.2	Workloads:	140
8.4	Modeling the performance of the algorithms	141
8.4.1	The three phases of atomic broadcast.	141
8.4.2	Wide-area network with three locations.	142
8.4.3	Wide-area network with two locations.	143
8.5	Experimental performance evaluation	144
8.5.1	Evaluation environments	145
8.5.2	Validation of the model with the experimental results	146
8.5.3	Comparing the performance of the three algorithms	148
8.6	Discussion	150

9 On the scalability of atomic broadcast algorithms 153

9.1	System model	156
9.1.1	System model, consensus and atomic broadcast	156
9.2	Scalable atomic broadcast	158
9.2.1	Presentation of the algorithm	158
9.2.2	Benefits and drawbacks of the scalable algorithm	160
9.2.3	Proof of correctness	161
9.2.4	Optimizations	162
9.3	Performance evaluation	164
9.3.1	Performance metrics, workload and the implementation framework	164
9.3.2	Evaluation environment	166
9.3.3	Results of the performance measurements	168

9.4	Discussion	178
10	Conclusion	181
10.1	Research assessment	181
10.1.1	Atomic broadcast	181
10.1.2	Experimental performance evaluations	183
10.2	Open questions and future research directions	184
A	Agreement algorithms	197
A.1	Reliable broadcast	197
A.2	Consensus	198
A.2.1	Chandra-Toueg consensus	198
A.2.2	Mostéfaoui-Raynal consensus	200
A.3	Atomic broadcast	202
A.3.1	Chandra-Toueg atomic broadcast	202
A.3.2	Moving sequencer uniform atomic broadcast	204
B	Modeling and validating the performance of atomic broadcast algorithms in high latency networks	211
B.1	Analytical performance in a wide area network with three locations	211
B.1.1	Chandra-Toueg atomic broadcast	211
B.1.2	<i>TokenFD</i> atomic broadcast	213
B.2	Analytical performance in a wide area network with two lo- cations	216
B.2.1	Chandra-Toueg atomic broadcast	217
B.2.2	<i>TokenFD</i> atomic broadcast	221

Contents

List of Figures

3.1	Example of two rounds r and $r + 1$ in the Heard-Of model, from the point of view of a process p_1	28
4.1	Example execution of the token based consensus algorithm	41
4.2	The Neko simulation model	48
4.3	Latency vs. throughput with a <i>normal-steady</i> faultload, $n = 3$ correct processes	50
4.4	Latency vs. throughput with a <i>crash-steady</i> faultload, one crashed process ($n = 3$ processes)	50
4.5	Latency overhead vs. throughput with a <i>crash-transient</i> faultload, one crash, $T_D = 0ms$, $n = 3$ processes	51
4.6	Latency overhead vs. throughput with a <i>crash-transient</i> faultload, one crash, $T_D = 100ms$, $n = 3$ processes	51
4.7	Latency vs. mistake recurrence time T_{MR} with a <i>suspicion-steady</i> faultload, $n = 3$ processes, low throughput	52
4.8	Latency vs. mistake recurrence time T_{MR} with a <i>suspicion-steady</i> faultload, $n = 3$ processes, high throughput	52
4.9	Latency vs. throughput with a <i>normal-steady</i> faultload, $n = 5$ (CT, MR) and $n = 7$ ($TokenFD$) correct processes	54
4.10	Latency vs. throughput with a <i>crash-steady</i> faultload, two crashed processes, $n = 5$ (CT, MR) or $n = 7$ ($TokenFD$)	54
4.11	Latency overhead vs. throughput with a <i>crash-transient</i> faultload, two crashes, detection time $T_D = 0ms$	55
4.12	Latency overhead vs. throughput with a <i>crash-transient</i> faultload, two crashes, detection time $T_D = 100ms$	55
4.13	Latency vs. mistake recurrence time T_{MR} with a <i>suspicion-steady</i> faultload, $n = 5$ (CT, MR) and $n = 7$ ($TokenFD$) processes, low throughput	56
4.14	Latency vs. mistake recurrence time T_{MR} with a <i>suspicion-steady</i> faultload, $n = 5$ (CT, MR) and $n = 7$ ($TokenFD$) processes, high throughput	56
4.15	Illustration of Lemma 4.2.2	62
5.1	Latency vs. message size in a system with 3 processes	73
5.2	Latency vs. message size in a system with 5 processes	73

List of Figures

5.3	Illustration of the violation of the <i>Validity</i> of atomic broadcast if consensus is executed directly on message identifiers	75
5.4	Intersection of the estimates received by two processes p and q ($n = 7$ processes and $f = 2$)	85
5.5	Latency vs. throughput of indirect consensus or (faulty) consensus	89
5.6	Latency vs. payload of indirect consensus or (faulty) consensus	89
5.7	Latency vs. payload of indirect consensus or consensus and uniform reliable broadcast. Reliable broadcast uses $O(n^2)$ messages.	92
5.8	Latency vs. payload of indirect consensus or consensus and uniform reliable broadcast. Reliable broadcast uses $O(n)$ messages.	92
5.9	Latency vs. throughput of indirect consensus or consensus and uniform reliable broadcast	93
6.1	The robust TCP protocol in the OSI reference model.	100
6.2	Lifetime of session and transport layer connections.	101
6.3	Opening phase of robust TCP	103
6.4	Reconnection phase of robust TCP	103
6.5	Structure of robust TCP control messages.	106
6.6	Results of the TCP Stream benchmark.	109
7.1	Quality of service model of a failure detector in the <i>suspicion-steady</i> faultload.	117
7.2	Latency vs. throughput with a <i>normal-steady</i> faultload	118
7.3	Latency vs. mistake recurrence time T_{MR} with a <i>suspicion-steady</i> faultload, $n = 3$ processes	120
7.4	Latency vs. mistake recurrence time T_{MR} with a <i>suspicion-steady</i> faultload, $n = 5$ (CT, MR) or $n = 7$ ($TokenFD$) processes	121
7.5	Communication pattern of CT in a run without (top) and with (bottom) a wrong suspicion.	121
7.6	Communication pattern of MR in a run without (top) and with (bottom) a wrong suspicion.	122
7.7	Latency vs. mistake duration T_M with a <i>suspicion-steady</i> faultload in a system with a mistake recurrence time of $100ms$. . .	124
7.8	Latency vs. throughput with a <i>normal-steady</i> faultload	126
7.9	Communication patterns of $TokenFD$ and $MovingSeq$ in good runs.	127
7.10	Throughput vs. offered load of the $TokenFD$ algorithm with a <i>normal-steady</i> faultload	128
7.11	Early latency vs. Mistake recurrence time T_{MR} of a group membership and a failure detector based algorithm, simulation results from [USS03]	129

8.1	Communication pattern of the Chandra-Toueg and <i>TokenFD</i> atomic broadcast algorithms in good runs.	139
8.2	Theoretical model of a wide area network with two or three locations	142
8.3	Wide area network evaluation environments in decreasing order of round trip times.	145
8.4	Latency vs. throughput of <i>CT</i> , <i>MR</i> and <i>TokenFD</i> in the WAN Three Locations setting.	147
8.5	Latency vs. throughput of <i>CT</i> , <i>MR</i> and <i>TokenFD</i> in the WAN 295 setting.	147
8.6	Latency vs. throughput of <i>CT</i> , <i>MR</i> and <i>TokenFD</i> in the WAN 20.1 setting.	149
8.7	Latency vs. throughput of <i>CT</i> , <i>MR</i> and <i>TokenFD</i> in the WAN 3.9 setting.	149
9.1	Execution of the scalable atomic broadcast algorithm.	159
9.2	Geographical distribution of the sites in the Grid'5000 WAN setup.	167
9.3	Latency vs. throughput of the Chandra-Toueg and <i>TokenFD</i> algorithms in a LAN	169
9.4	Latency vs. throughput of the three algorithms for system sizes of 7, 13 and 21 processes in a LAN	169
9.5	Latency vs. throughput of the scalable atomic broadcast algorithm with a kernel of 3 processes in the LAN.	170
9.6	Latency vs. throughput of the scalable atomic broadcast algorithm with a kernel of 7 processes in the LAN.	170
9.7	Latency vs. system size of the original and scalable atomic broadcast algorithms in the LAN setup.	172
9.8	Latency vs. throughput of the Chandra-Toueg and <i>TokenFD</i> algorithms in the WAN setup.	174
9.9	Latency vs. throughput of the three algorithms for system sizes of 3, 15 and 23 processes in WAN setup.	174
9.10	Latency vs. throughput of the scalable atomic broadcast algorithm with a kernel of 3 processes in the WAN setup.	176
9.11	Latency vs. throughput of the scalable atomic broadcast algorithm with a kernel of 7 processes in the WAN setup.	176
9.12	Latency vs. system size of the original and scalable atomic broadcast algorithms in the WAN setup.	177
A.1	Communication pattern of the Chandra-Toueg $\diamond S$ consensus algorithm in good runs.	198
A.2	Communication pattern of the Mostéfaoui-Raynal $\diamond S$ consensus algorithm in good runs.	200

List of Figures

A.3	Communication pattern of the Chandra-Toueg atomic broadcast algorithm ($n = 3$ processes).	202
A.4	Communication pattern of the Moving Sequencer atomic broadcast algorithm ($n = 5$ processes).	206
B.1	Execution pattern of the Chandra-Toueg consensus algorithm in the two-location wide area network model.	219
B.2	Coordinator processes for <i>abroadcast</i> messages.	220
B.3	Token circulation in the two-location wide area network model and a presentation of which token contains the messages <i>abroadcast</i> by the three processes.	221

List of Tables

4.1	Summary of the different approaches that ensure the <i>Uniform agreement</i> property of atomic broadcast	64
4.2	Illustration of which processes send and receive messages in a given round of the token based consensus algorithm in the Heard-Of model	67
6.1	Java vs. Robust TCP in the three benchmarks.	108
8.1	Example of the average latency to <i>adeliver</i> a message in the two-location wide area network model	144
B.1	Average latency to <i>adeliver</i> a message in the three-location wide area network model, using Chandra-Toueg’s algorithm (with <i>CT</i> and <i>MR</i> ’s consensus algorithm and a fixed initial coordinator).	212
B.2	$CostSend_{i,j}$, $CostWait_{i,j}$, $CostOrder_j$ and $OrderedBy_{i,j}$ in the three-location wide area network model, using Chandra-Toueg’s algorithm (with <i>CT</i> ’s consensus algorithm and a shifting initial coordinator).	214
B.3	$CostSend_{i,j}$, $CostWait_{i,j}$, $CostOrder_j$ and $OrderedBy_{i,j}$ in the three-location wide area network model, using Chandra-Toueg’s algorithm (with <i>MR</i> ’s consensus algorithm and a shifting initial coordinator).	214
B.4	$CostSend_{i,j}$, $CostWait_{i,j}$, $CostOrder_j$ and $OrderedBy_{i,j}$ in the three-location wide area network model, using the <i>TokenFD</i> atomic broadcast algorithm.	215
B.5	Average latency to <i>adeliver</i> a message in the two-location wide area network model, using Chandra-Toueg’s algorithm (with <i>CT</i> or <i>MR</i> ’s consensus algorithm) or the <i>TokenFD</i> algorithm. Results are given for an initial coordinator (<i>MR</i> , <i>CT</i>) on a local location, on the distant location or that shifts at each new consensus execution.	216

List of Tables

Introduction

1.1 Research context and motivation

The need for dependable computing Within only a couple of generations, the so-called digital revolution has taken the world by storm: today, almost all human beings interact, directly or indirectly, at some point in their life, with a computer system. Computers are present on our desks, computer systems control the antilock braking system and the stability control in cars, they collect usage statistics in elevators in order to anticipate maintenance and repair operations. Computer systems also operate critical systems, such as nuclear power plants, airplane control systems [TLS05] or space rockets [Lan97]. Furthermore, computer systems are not only omnipresent, but also increasingly networked.

As the use of computer systems has increased dramatically over the past decades, the needs and expectations associated with these systems have also increased. In the case of critical systems, dependability has always been a major concern, as the price of a system failure is high, in terms of human lives or economical loss. However, even in the case of non-critical systems, one of the crucial properties is availability [Hen99] (*i.e.* the fraction of the time during which the system provides a service to the users): the costs and negative publicity of a system outage (be it a commercial web site or a stock exchange for example) are often considerable.

High availability is achieved in two ways: (1) by increasing the time a system is up¹ (by avoiding that the system ever goes down) or (2) by reducing the recovery time in case it is down. The second approach is difficult or even impossible to apply to critical systems where the slightest interruption of service is unacceptable.

In the first approach, several system designs are possible. The first design choice is to avoid system failures (*fault avoidance*), while the second one is to tolerate and hide failures within the system (*fault tolerance*). These

¹When a system provides its service, we say that it is *up*. Otherwise, it is *down*.

designs are of course complementary and can be combined [TLS05]. The second design is however appealing, as it can be implemented by replication: instead of having a single machine (a *replica*) providing a service, the system is composed of several replicas running the service. Thus, if one of the replicas fails, the service is still provided by the remaining replicas. Furthermore, replication is interesting as it can be achieved by using software running on commodity hardware, thus avoiding the high cost of special purpose hardware.

Dependability through replication Replication, although intuitive to understand, is complex to implement in practice, as several system components (the *replicas*) have to interact in order to ensure the consistency of the system as a whole. Since the replication is mostly transparent to the user of the service (*i.e.*, the user should not notice that several replicas, rather than a single machine, handle its request), the replicas must coordinate their actions and replies to a user request. This coordination contributes to the complexity of replication.

The complexity depends on factors such as the degree of consistency between replicas or the amount of shared state. A strong consistency isn't necessary, for example, between two servers providing a domain name service or Google's web page indexing, and thus requires less complex replication techniques. The most implemented replicated services in practice are precisely those with relatively weak consistency requirements.

On the other hand, systems with stronger consistency between replicas are less deployed in practice. Replication techniques, such as quorum systems and group communication (atomic broadcast in particular), provide services that simplify the replication problem, by hiding issues such as the communication between the processes, process crashes and synchrony.

Group communication: hiding the complexity of replication Group communication [HT94] provides primitives that allow one-to-many and many-to-many communication between a set of replicas, called *processes*. The primitives address problems such as reliable delivery of messages, total order of delivery on all processes and resilience to process failures. An agreement problem is at the center of most primitives: the group of processes have to *agree* on the outcome of some operation, in order to fulfill the specification of the primitives.

Agreement problems are difficult to solve. If no assumptions are made on the relative speed of different components in the system for example, some agreement problems are impossible to solve [FLP85]. Atomic broadcast – a group communication primitive at the heart of this work – allows processes to broadcast messages to the group and then deliver them in the same total order, even if processes broadcast messages quasi simul-

taneously. Atomic broadcast is especially useful for replication: since all replicas deliver messages in the same order, their state is kept consistent (assuming that the state evolves deterministically as a function of the delivered messages).

Different approaches for solving atomic broadcast Atomic broadcast is one of the agreement problems that is impossible to solve in an *asynchronous* system (without any assumptions on the relative speed of the processes or the network) when processes may crash [FLP85, CT96]. In order to circumvent the impossibility of atomic broadcast (and consensus), the asynchronous system can be extended with oracles that provide some additional information about the state of the system.

One such oracle is the *unreliable failure detector* [CT96], which provides (unreliable) information about process failures and thus allows previously impossible agreement problems to be solved [CHT96]. Furthermore, any algorithm that uses a failure detector is *directly* exposed to process failures and suspicions (unlike algorithms which delegate the handling of suspected process failures to a separate module providing a group membership service, for example). As a consequence, such algorithms are generally designed to tolerate wrong suspicions (*i.e.* when a process is incorrectly suspected of having crashed), since the unreliable nature of the failure detectors implies that suspicions shouldn't generate too costly operations.

Atomic broadcast algorithms can also be classified following the ordering mechanism that they use [DSU04]. In the *fixed sequencer* approach, for example, a single process is responsible for ordering messages, whereas in a *destinations agreement* approach, the destination processes of a message m agree on the order of delivery of m with respect to the other broadcast messages. Other ordering mechanisms include the *moving sequencer* or *privilege based* approaches which are both related to token based algorithms and discussed below.

Token based ordering mechanisms implicitly provide a flow control that limits the contention on the network and the processes, and thus allows high system loads. A token can be used in several ways: in [Cri91] the token aggregates the ordered messages. The token can also limit the access to the network: in [AMMS⁺95, CMA97], only the token holder is allowed to send messages on the network (and the messages are thus easily ordered, since a single sender exists at a given point in time). Finally, the token can be used to select a sequencer process [CM84, MS01, WMK94] that receives the messages from all other processes and assigns sequence numbers. Since the token circulates, the load of assigning sequence numbers to messages is distributed among the processes in the system.

1.2 Research contributions

Replication: Understanding the advantage of atomic broadcast over quorum systems In the context of software replication for fault-tolerant services, several techniques have been presented. Quorum systems was the first of these techniques to be introduced and since then, the understanding of problems related to replication has made significant progress. An important step was the introduction of *group communication* (and atomic broadcast) which defines a middleware layer that hides most of the hard problems related to replication.

Quorum systems have, after a decrease of interest, recently garnered renewed attention in the context of Byzantine faults. Additionally, questions have arisen on the advantages of group communication over quorum systems for replication. We attempt to clarify this issue, and point out precisely when and why group communication is a better solution than quorum systems.

A token based atomic broadcast algorithm As mentioned previously, atomic broadcast allows processes, organized in a group, to broadcast messages and deliver them in the same total order. A large number of atomic broadcast algorithms have been presented in the literature. Several approaches exist (1) to tolerate failures and (2) to order messages.

We present a new algorithm that uses a token based mechanism (to order messages) and unreliable failure detectors (to tolerate failures). The goal of the algorithm is to benefit from the characteristics generally associated with failure detector based algorithms (the low cost of wrong suspicions) and token based algorithms (efficiency in terms of throughput, due to the natural flow control provided by the token). The performance of our token based algorithm is simulated and compared to two other failure detector based algorithms. The simulation shows that the new token based algorithm achieves lower latencies and a higher throughput than both other algorithms in most settings.

Solving atomic broadcast with indirect consensus Atomic broadcast can be reduced to consensus, as shown by Chandra and Toueg [CT96]. In this reduction, consensus is run on sets of messages, in order to determine the delivery order of those messages. While this is correct from a theoretical point of view, it is inefficient in practice, especially if messages are large. Instead, if consensus is executed on message identifiers, the messages themselves only need to be diffused once and the ordering process is done on light-weight message identifiers.

Executing consensus on message identifiers has always been seen as being easy, given a consensus algorithm on messages. We show that this is

not necessarily the case: if at least one process can crash, then Chandra and Toueg's reduction of atomic broadcast to consensus on message identifiers can lead to a faulty execution. We address this issue and show how to adapt the atomic broadcast and consensus algorithms. The modifications are not trivial for all consensus algorithms and can affect their resilience.

Robust TCP connections: implementing quasi-reliable channels A number of algorithms in fault-tolerant computing assume so-called quasi-reliable channels: if a process p sends a message m to another process q , then q eventually receives m if neither p nor q fail. An obvious way to implement these channels is to use TCP connections, which hides most problems of the communication channel from the programmer: message loss, duplicates and short losses of connectivity. Unfortunately, TCP does not adequately handle link failures: TCP breaks the connection if connectivity is lost for some duration.

Robust TCP connections address this problem: they present the same interface as regular TCP connections, but never break due to network problems. We define a session-layer protocol on top of TCP that ensures possible reconnections and provides exactly-once delivery for all transmitted data. A prototype, implemented in Java, has less than 10% overhead over TCP sockets with respect to response time and throughput.

Comparing atomic broadcast algorithms in a local area network

The first approach in evaluating the performance of distributed algorithms is by simulation or by analyzing the message complexity and time complexity. This, however, only gives partial information on the performance of an algorithm once it is deployed in a real environment. To complete the performance analysis, it is necessary to experimentally evaluate the algorithm in its target deployment environment.

The first environment that is considered is a local area network. In this setting, we evaluate the new token based algorithm (noted *TokenFD*), as well as two failure detector based algorithms and another token based algorithm that uses a group membership service to tolerate failures.

This experimental evaluation confirms that among the failure detector based algorithms, *TokenFD* achieves the highest throughput when neither failures nor suspicions occur. Secondly, when wrong suspicions occur frequently, the latency of *TokenFD* is less affected than the latency of the two other algorithms using failure detectors. Finally, compared to the algorithm using a group membership service, *TokenFD* cannot sustain as high a throughput in good runs, *i.e.* runs without failures nor suspicions. However, when wrong suspicions occur, the group membership service performs costly operations that are not necessary when failure detectors are used, as in the case of *TokenFD*.

Evaluating the performance of atomic broadcast algorithms in high latency networks The second setting in which we evaluate atomic broadcast algorithms is wide area networks. We start by modeling the performance of the three algorithms using failure detectors in a simple system model with three processes, distributed on two or three different locations.

We then experimentally evaluate the algorithms in four different wide area networks, with varying round trip times. The experiments show that for moderate throughputs, the modeled performance is close to the experimental results. Furthermore, we also show the main characteristic that affects the performance of an atomic broadcast algorithm in a wide area network is the number of communication steps it needs, rather than the number of messages it sends. This result remains true, whether a network with large (295 *ms*) or small (4 *ms*) round trip times is considered.

On the scalability of atomic broadcast algorithms The final experimental evaluation in this thesis examines the impact of the size of the system on the performance of the three atomic broadcast algorithms using failure detectors. The size of the system varies between 3 and 23 processes, running on a local area network or on a wide area network with seven different locations.

The performance evaluation shows that the latency of all three algorithms is strongly affected by the size of the system. As a consequence, we present an algorithm that limits the actual ordering algorithm to a kernel, *i.e.* a subset of the system, and only sends updates to the other processes. We show that the performance of this *scalable* algorithm (with the three ordering algorithms running only on the kernel) is hardly affected by the size of the system and instead depends on the size of the kernel. Thus, the kernel can be tailored to suit the fault tolerance needs of the application, and additional processes (that act as a cache of the distributed state) can be added to the system at a low cost.

1.3 Structure of the thesis

Preliminaries Chapter 2 discusses the advantages of using group communication instead of quorum systems to implement replicated services. Chapter 3 starts the first part of the thesis by introducing the model of distributed systems that we consider and defines concepts such as faulty components or communication channel types. This chapter also includes formal descriptions of the problems that are considered throughout this thesis.

Solving atomic broadcast Chapter 4 presents a novel atomic broadcast algorithm that uses a token to order messages and an unreliable failure detector to tolerate process failures. The chapter also includes a discussion on possible optimizations of the algorithm, as well as a presentation of the algorithm in a different system model.

Chapter 5 discusses the reduction of atomic broadcast to a sequence of consensus executions on message identifiers. This reduction is not always trivial: we show examples of a consensus algorithm that is easily transformed to handle message identifiers, and an algorithm whose resilience is affected by the transformation.

Performance evaluations The second part of the thesis presents several experimental performance evaluations of atomic broadcast algorithms. Chapter 6 proposes an implementation of quasi-reliable channels, based on TCP. Chapter 7 compares four atomic broadcast algorithms in a local area network. The performance of the algorithms is examined in systems where no failures or failure suspicions occur, and in the case of frequent wrong suspicions. Chapter 8 examines the performance of three failure detector based atomic broadcast algorithms in several wide area networks. A simple model for the performance of the algorithms is presented first and later validated by the experimental results. Finally, Chapter 9 examines the impact of the system size on the three same failure detector based algorithms, in local and wide area networks. We also present a scalable algorithm that limits the ordering of messages to a fixed subset of the system.

Conclusion Finally, the main contributions of this work are summarized in Chapter 10 and future research directions are discussed.

Replication: understanding the advantage of atomic broadcast over quorum systems

The requirement for highly reliable and available services has been continuously increasing in many domains for the last decade. Several approaches for designing fault-tolerant services exist. The focus of this chapter is on software replication. Replication allows a number of replicas to crash without affecting the availability of the service.

Quorum systems [Tho79, Gif79] was the first technique introduced to manage replication. Since this period, a lot of progress has been accomplished in the understanding of the problems related to replication. An important step has been the introduction of *group communication*, which defines a middleware layer that hides most of the hard problems related to replication [Sch03]. The advent of group communication has temporarily led to a decrease of interest in quorum systems. However, there has been recently a renewed interest in quorum systems for Byzantine faults [MR98], an issue not addressed previously. Moreover, there are now here and there people disagreeing on the advantage of group communication over quorum systems for replication. The goal of this chapter is to clarify this issue, and point out precisely when and why group communication is a better solution.

The rest of the chapter is organized as follows. Section 2.1 introduces our system model and three isolation degrees, a key issue to understand the respective scope of quorum systems and atomic broadcast. Section 2.2 discusses the absence of isolation requirements. Section 2.3 discusses the case when only read-write isolation is required. Section 2.4 discusses general isolation requirements. Finally, a discussion in Section 2.5 concludes

this chapter.

2.1 Different isolation degrees

In the context of replication, one of the key issues is the semantics that have to be provided. We consider in this chapter a finite set of processes, where each process issues a sequence of operations over a finite quantity of replicated data. Without restriction of generality, we consider that each operation is either a *read* or a *write*. A read operation reads one replicated data; a write operation writes one replicated data. The semantics define the result of each of these read and write operations. One key aspect of the semantics is the *isolation* property, as defined in the context of database systems [WV02]. We distinguish the following degrees of isolation:

No isolation Any interleaving of operations is possible; only the semantics of each individual read or write operation is defined.

Read-write isolation In addition to the individual semantics of read and write operations, a read followed by a write on the same data are executed in isolation.

General isolation In addition to the individual semantics of read and write operations, any sequence of operations can be executed in isolation.

To illustrate the three cases, consider two processes p, q , two replicated data X, Y , and the following sequences of operations:¹

- Sequence of operations issued by p : $r_p(X), w_p(X), r_p(Y), w_p(Y)$
- Sequence of operations issued by q : $r_q(Y), w_q(Y), r_q(X), w_q(X)$

With no isolation, any interleaving of operations of the two processes is possible.

We express isolation using $[\dots]$ brackets. Here is the same sequence of operations with read-write isolation (the consecutive read-write operations are executed in isolation):

- Sequence of operations issued by p : $[r_p(X), w_p(X)], [r_p(Y), w_p(Y)]$
- Sequence of operations issued by q : $[r_q(X), w_q(X)], [r_q(Y), w_q(Y)]$

Finally, general isolation allows us to specify for example the following isolation requirement:

- Sequence of operations issued by p : $[r_p(X), w_p(X), r_p(Y), w_p(Y)]$
- Sequence of operations issued by q : $[r_q(X), w_q(X), r_q(Y), w_q(Y)]$

¹ $r_p(X)$ (respectively $w_p(X)$) denotes a read (respectively a write) of data X by process p .

2.2 No isolation

Read-write operations with no isolation corresponds to the notion of *register* [Lam86]. The strongest register semantics, called *atomic register*, ensure that the read and write operations behave as if each operation op issued by process p happened instantaneously at some time $t \in [op_{start}, op_{end}]$, where op_{start} is the time at which the op is issued by process p , and op_{end} is the time at which op has completed on p [Lam86].

Atomic registers can be implemented in an asynchronous system (which is defined as a system in which there is no bound on the transmission delay of messages, nor on the relative speed of processes). Quorums are here well suited to implement atomic registers. As an example, consider the data X replicated on several servers X_i , where each server X_i manages (1) a copy of the data and (2) a version number. A quorum is defined as any subset of servers. Quorum systems distinguish *read* quorums and *write* quorums, which must satisfy the following properties [Gif79]:

- Any read quorum has a non-empty intersection with any write quorum.
- Any two write quorums have a non-empty intersection.

Let n be the number of replicas. One standard way to satisfy these properties is the following [Gif79]:

- A read quorum is any subset of servers of size $\lceil \frac{n+1}{2} \rceil$.
- A write quorum is also any subset of servers of size $\lceil \frac{n+1}{2} \rceil$.

The operation $w_p(X \leftarrow val)$ (write val to X) by p is performed as follows: (1) p reads the version number from a read quorum, (2) then the local variable vn is set to the highest version number read, and finally (3) the value val with version number $vn + 1$ is written to a write quorum.

The *read* operation $r_p(X)$ is slightly less intuitive: (1) the client reads the pair $(value, version)$ from a read quorum, (2) the read operation returns the value val with the highest version number, and finally (3) the value val is written to a write quorum.²

The specificity of this solution can be summarized as follows: (1) data is sent back and forth between the servers X_i and the client process p , and (2) servers only send and receive data. We will come back to this point later.

²Without (3), the atomic register semantics is not ensured. To see this, consider (a) $w_{p_1}(X \leftarrow w)$ by p_1 that starts at $t = 1$ and ends at $t = 6$, (b) a read operation $r_{p_2}(X)$ by p_2 that starts at $t = 2$, reads w , and terminates at $t = 3$, and (c) a read operation $r_{p_3}(X)$ by p_3 that starts at $t = 4$ and ends at $t = 5$. Without (3), p_3 could read an old value rather than w , which is required by the atomic register semantics.

2.3 "Read-write" isolation only

To show the limitations of the atomic register semantics, and the need for read-write isolation, consider the following sequence of operations, where process p wants to increment X , while process q wants to decrement X . If X is initially 0, then without read-write isolation, the following execution is possible:³

- $r_p(X \Rightarrow 0), r_q(X \Rightarrow 0), w_p(X \leftarrow 1), w_q(X \leftarrow -1)$

This execution is clearly not desired (the final value of X must be 0). A correct execution requires that p and q execute the read-write sequence in mutual exclusion, *i.e.*, in isolation. This can be expressed as follows, where E_{CS}/L_{CS} allows a process to enter/leave the critical section:

- Operations issued by p :⁴ $E_{CS}, r_p(X \rightarrow u), u \leftarrow u + 1, w_p(X \leftarrow u), L_{CS}$
- Operations issued by q : $E_{CS}, r_q(X \rightarrow u), u \leftarrow u - 1, w_q(X \leftarrow u), L_{CS}$

2.3.1 Read-write isolation and the consensus problem

We first show that read-write isolation cannot be solved in an asynchronous system with crash failures. Then we discuss the implementation of read-write isolation (1) with quorum systems and (2) with atomic broadcast (a group communication primitive).

Consensus is a well known problem defined over a finite set of processes, in which each process has an initial value and all processes that do not crash have to agree on a common value that is the initial value of one of the processes [CT96]. Consensus is not solvable in an asynchronous system if processes may crash [FLP85]. This impossibility also applies to read-write isolation; it follows directly from the fact that read-write isolation is powerful enough to solve consensus (see also [Her88, Her91]). To show this, consider consensus to be solved among n processes p_1, \dots, p_n , with val_i the initial value of process p_i . Let the data be here a vector V of $n + 1$ elements $V[0], \dots, V[n]$. Initially, we assume $V[0] = 0$, and all other elements $V[j]$ undefined. Each process p_i executes Algorithm 2.1, where $V[0]$ is incremented and $V[V[0]]$ written inside a critical section (lines 2-5).

If at least one process p_i is correct, then $V[1]$ is written (with the initial value of one of the processes). Moreover, since all processes decide on the value $V[1]$, they all decide the same value, which is the initial value of

³ $r_p(X \Rightarrow v)$ denotes a read operation that returns the value v .

⁴ $r_p(X \rightarrow u)$ denotes that the value returned by the read operation is stored into the local variable u .

Algorithm 2.1: Solving consensus with read-write isolation (code of process p_i)

1: E_{CS}	{Enter Critical Section}
2: $r_{p_i}(V \rightarrow u)$	{Read vector V into local vector u }
3: $u[0] \leftarrow u[0] + 1$	
4: $u[u[0]] \leftarrow val_i$	
5: $w_{p_i}(V \leftarrow u)$	{Write u to vector V }
6: L_{CS}	{Leave Critical Section}
7: $r_{p_i}(V \rightarrow u)$	{Read vector V into local vector u }
8: decide $u[1]$	{Consensus decision}

one of the processes. So read-write isolation allows us to solve consensus, which shows the contradiction, *i.e.*, read-write isolation cannot be implemented in an asynchronous system with process crashes.

2.3.2 Implementing read-write isolation with quorums

Since read-write isolation cannot be implemented in an asynchronous system with process crashes, we need additional assumptions. The quorum solution of Section 2.2 can be extended to provide read-write isolation if we can solve the mutual exclusion problem. Implementing mutual exclusion requires to handle the following situation:

- Process p executes E_{CS} and gets permission to enter the critical section.
- Process p crashes before leaving the critical section.

In this case, p will never release the critical section, *i.e.*, the critical section must be released on behalf of p . This requires a crash detection mechanism that detects the crash of p if and only if p has crashed (the critical section must be released if and only if p has crashed). This corresponds to a *perfect failure detector* [CT96], which is a strong requirement. Note that in addition to a perfect failure detector, if the read/write quorums are defined as in Section 2.2, the solution also requires a majority of correct processes (to always ensure the existence of a read quorum and of a write quorum).

2.3.3 Implementing read-write isolation with atomic broadcast

We now describe a different solution to read-write isolation, which uses a group communication primitive, namely *atomic broadcast* (also called *total order broadcast*). Atomic broadcast allows to broadcast messages to a group of processes, while ensuring that messages are delivered by all members of

Algorithm 2.2: Model for read-write isolation (code of process p)

1: E_{CS}	{Enter Critical Section}
2: $r_p(X \rightarrow u)$	{Read X into local variable u }
3: $u \leftarrow f(u)$	{Update u }
4: $w_p(X \leftarrow u)$	{Write u to X }
5: L_{CS}	{Leave Critical Section}

the group in the same order. A formal definition can be found in Chapter 3. To show the implementation of read-write isolation with atomic broadcast, we model the execution of each process as shown in Algorithm 2.2.

Process p first reads X into a local variable u , then does some local computing expressed by the function $f(u)$, and finally writes the new value of u to X .

With atomic broadcast, denoted by $abroadcast()$, the above schema can be implemented as follows, using a technique called state machine approach [Lam78, Sch93a]. The technique distinguishes between (1) the code of process p (Algorithm 2.3) and (2) the code of a server X_i that manages a copy x_i of the data X (Algorithm 2.4).

Algorithm 2.3: Read-write isolation (code of process p)

1: $abroadcast(f)$ to g_X	{ g_X is the group of servers X_i }
2: wait to receive <i>done</i> from at least one server X_i	

Algorithm 2.4: Read-write isolation (code of server X_i)

1: loop	
2: wait for the <i>adelivery</i> of f sent by some process p	
3: $x_i \leftarrow f(x_i)$	{ x_i is the local copy of X managed by server X_i }
4: send(<i>done</i>) to p	

Every server X_i receives the update functions f in the same order, and updates its copy x_i using the same update function. Moreover, each server x_i executes one update function before considering the next one. So Algorithms 2.3 and 2.4 correctly implement atomic registers with read-write isolation. Indeed, the solution requires to solve atomic broadcast. Atomic broadcast is solvable in an asynchronous system augmented with the failure detector $\diamond S$ in a group g_X ,⁵ and a majority of correct servers [CT96].

⁵ $\diamond S$ satisfies the following properties: (1) Eventually every process in g_X that crashes is permanently suspected by every correct process in g_X , and (2) there is a time after which some correct process in g_X is never suspected by any correct process in g_X [CT96].

2.3.4 Discussion

If we compare the requirements of the quorum solution and of the atomic broadcast solution, we observe the following. The two solutions require a majority of correct processes; the quorum solution requires a perfect failure detector, whereas the atomic broadcast solution only requires the weaker failure detector $\diamond S$ (see [CT96] for a comparison of failure detectors). To understand how much $\diamond S$ is weaker than a perfect failure detector, note that $\diamond S$ allows an *unbounded* number of false crash suspicions, while a perfect failure detector does not allow a *single* false suspicion.

What makes the difference? *In the quorum solution, the update function f is executed by the client process itself. In the atomic broadcast solution, the update function f is executed by the servers.* The former solution requires (1) mutual exclusion, and (2) to send data back and forth between the client and the servers. The atomic broadcast solution requires only to send the update function f to the servers. Executing f on the servers is a more clever solution than executing f on the client!

2.4 General isolation

We now discuss the implementation of general isolation. The quorum solution can trivially be extended to handle general isolation. Indeed, whether mutual exclusion protects two operations or more than two operations does not make a difference.

Can the atomic broadcast solution be extended to handle general isolation? Yes, if specific conditions are met (which also means that the solution is not always applicable):

- when the update function f can be defined statically, *e.g.*, when the application can be implemented using stored procedures, and
- when the identity of the servers to which f must be sent is statically known.

The atomic broadcast solution may also require atomic broadcasts to multiple groups [GS01]. We now give two examples where these conditions are satisfied.

Example 1: Consider two replicated data X and Y , representing two bank accounts. Assume a user that wants to withdraw an amount w from account X and deposit w on the account Y . This can be expressed by the following update function:

$$f \equiv (\text{sub}(X, w); \text{add}(Y, w))$$

The user simply issues *abroadcast*(f) to $g_X \cup g_Y$, where g_X , respectively g_Y , are the group of replicas of X , respectively Y . Upon delivery of f , a server X_i decrements its local copy x_i by w , and a server Y_i increments its local copy y_i by w .

Example 2: Let us modify slightly Example 1, such that the transfer of w from account X to account Y takes place if and only if $X \geq w$. This can be expressed as follows:

$$f \equiv (\text{if } X \geq w \text{ then } \text{sub}(X, w); \text{add}(Y, w) \text{ endif})$$

This leads to the following problem: while a server x_i can evaluate the condition $X \geq w$, a server y_i cannot. Nevertheless, this case can still be implemented using atomic broadcast: each server x_i after the evaluation of the condition $X \geq w$, sends *true* or *false* to the servers in g_Y . A server in g_Y waits for this message to know whether or not to execute the *add* operation.

In these two examples the set of data to be accessed is known statically. If this condition is not met, which is quite common in the case of database transactions, then the atomic broadcast solution cannot be used (since it cannot be known to which servers to send the update function). Note that the function could be sent to *all* servers, but this solution might be too costly or even impossible to implement.

2.5 Discussion

There is a common misunderstanding of the advantage of group communication over quorum systems to manage replicated data. We have tried to clarify this issue by showing the basic difference between the two techniques: when isolation needs to be provided, *group communication consists in sending the update function to the data servers, while with quorum systems servers send the data to the clients where the update function is performed*. The first solution requires weaker extensions to the asynchronous system, and so has obvious advantages. We have also shown that the use of group communication is not restricted to read-write isolation. This contradicts the claim of Cheriton and Skeen in [CS93] in the context of the CATOCS controversy,⁶ where they write that *CATOCS cannot ensure serializable ordering between operations that correspond to a group of messages (...)* *Locking is the standard solution*.⁷ As shown, this argument is not correct. Apart from this specific issue, we believe that this chapter should allow in the future to clearly understand the merits of group communication over quorum systems to manage replication. It also underlines the relevance of using group

⁶CATOCS = Causally and totally ordered communication support.

⁷Note that atomic broadcast can be used for locking

communication (as opposed to other techniques) to achieve fault tolerance. This thus justifies the study of atomic broadcast in the following chapters.

Part I

Solving Atomic Broadcast with Failure Detectors and Token based Algorithms

System models and definitions

3.1 System models

We consider a distributed system where a set of n processes interact by passing messages on a communication subsystem. The set of processes is noted $\Pi = \{p_0, \dots, p_{n-1}\}$. These processes all have their own memory space and do not have access to any shared memory. Furthermore, we assume that there is an implicit order on the processes and the k^{th} successor of a process p_i is $p_{(i+k) \bmod n}$ (which is, from now on, simply noted p_{i+k} for the sake of clarity).

We consider that each pair of processes in the system is connected by a point-to-point communication channel. This point-to-point model is basic and can be emulated in other network models. All the messages sent on the network are unique (and are identified, for example, by the identifier of the sender process and a sequence number) and taken from a set \mathcal{M} .

This system definition is general and needs to be refined in order to be useful for developing distributed algorithms. The rest of this chapter thus specifies the characteristics of the different elements of the system. We now discuss the two main characteristics of the system: the *failure modes* and *synchrony*.

3.1.1 Failure modes

The work in this thesis is centered around dependability and fault-tolerance. The distributed systems considered thus naturally allow components, be it processes or communication channels, to fail. The behavior of faulty components is important when solving problems in a distributed system. The following paragraphs specify what failure modes we consider, *i.e.* how processes are expected to fail, as well as the reliability properties of the

considered communication channels.

3.1.1.A Process failures

The general classes of process failures considered in distributed systems are the following:

Fail-stop Upon failing, a process stops executing any further computation step and does not send or receive any further message. Once a process has crashed it never recovers. This type of failure is also referred to as *crash* failures.

Omission A faulty process can omit to send or receive some of the messages and still continue executing.

Byzantine The byzantine failure class is the most general. Indeed, a byzantine process may behave arbitrarily: it can alter messages, omit to execute send or receive actions, create spurious messages or exhibit any other malicious behavior. Byzantine failures are not supported by the algorithms presented in this thesis and are consequently not discussed any further.

Most of the algorithms presented in this thesis assume that processes are only affected by fail-stop faults [SS83]. Omission failures are shortly considered by the algorithm presented in Section 4.3.

A process that never crashes is said to be *correct*, otherwise it is *faulty*. Furthermore, a process that crashes is faulty even before the time of the crash: the correct/faulty property of a process is tied to the entire execution of the system. Out of the n processes in the system, at most f may be faulty. The value of f depends on n , on the failure type and on the algorithm that is considered.

The fail-stop failure model is interesting for several reasons. Despite its apparent simplicity, it affects the solvability of problems, as discussed later in the context of system synchrony (Section 3.1.2). Furthermore, the failure model often matches the behavior of processes in real-world systems (if we exclude systems subject to malicious attacks). Indeed, errors that occur in a real environment, such as memory and message corruption, can be resolved by error correction techniques or by redundant executions of the same code [Avi85, Ran75]. Other incidents, such as power outages, cause behaviors that are close to the fail-stop model. Finally, programming techniques, such as the frequent use of assertions, limit the risk of processes behaving arbitrarily.

Experimental studies in which errors are injected into distributed systems, such as [CC98] and [MBK⁺05], confirm that a fail-stop behavior is

often observed. The authors also propose slight modifications to the systems that increase the proportion of errors that lead to clean crashes, rather than arbitrary behaviors.

3.1.1.B Communication channels

We now present the reliability properties of the communication channels that define the network subsystem. The communication channels are defined by the following properties:

No creation If a process q receives a message m from another process p , then p sent m to q .

No duplication A process q receives a given message m at most once.

No loss If a correct process p sends a message m to a correct process q , then q eventually receives m .

Fair loss If a correct process p sends a message m infinitely often to a correct process q , then q receives m an infinite number of times. The *Fair loss* property is weaker than *No loss*: if a channel satisfies the *No loss* property, it also satisfies the *Fair loss* property.

The algorithms in this thesis require one of the two following types of channels (that are defined by three of the four properties above).

Quasi-reliable channels are used by most of the algorithms presented and studied in this thesis. This type of channels is defined by the *No creation*, *No duplication* and *No loss* properties.

Fair-lossy The fair-lossy channel is defined by the *No creation*, *No duplication* and *Fair loss* properties.

The quasi-reliable channels are needed by the algorithms defined in the asynchronous system model augmented with unreliable failure detectors (see Section 3.1.2.D). The fair-lossy channels are used in the context of the Heard-Of system model, presented in Section 3.1.3.

3.1.2 Synchrony

The definition of a system presented above gives the elements (processes and communication channels) that constitute the distributed system, as well as their expected behavior in the case of failures. The timing assumptions on the operations of the processes and channels are defined by the *synchrony* of the system.

More specifically, the synchrony addresses two main characteristics of a distributed system: the ratio between the processing speeds of the different processes and the time needed to transmit a message on the network.

The synchrony of a system is important as it has a direct impact on the set of problems that are solvable in that system. A system \mathcal{S} with strong synchrony constraints allows more problems to be solved than a system \mathcal{S}' with weaker synchrony. Conversely, an algorithm that solves a problem in a system \mathcal{S}' also solves the problem in a system \mathcal{S} with stronger synchrony.

The following paragraphs present the asynchronous and synchronous system models, the two extremes of the synchrony spectrum. Models with intermediate synchrony characteristics are then also presented.

3.1.2.A Asynchronous system

In an *asynchronous* system, there are no assumptions on the relative speed of processes or the time needed to transmit a message on the network [Sch93b]. As mentioned above, this minimal synchrony has two consequences: (1) the set of problems that can be solved in this model is smaller than the same set in any other model and (2) algorithms designed for the asynchronous model are implementable in virtually any real system, since such a real system has stronger synchrony characteristics than the asynchronous system model.

The first consequence implies that a large number of distributed problems are not solvable in the asynchronous model in the presence of faulty processes [Lyn89, FR03], such as for example consensus [FLP85] and atomic broadcast (presented in Section 3.2), terminating reliable broadcast [Lar03, FRT99], non-blocking atomic commitment [BT93, CB03] or termination detection [MFVP05]. Only the class of *easy* problems, following the terminology in [FRT99], are solvable in an asynchronous system with process crashes (such as for example the reliable broadcast problem, presented below in Section 3.2.2).

However, following the second consequence above, an algorithm that solves a problem in the asynchronous model can be implemented in almost any kind of real system, ranging from a group of identical computers on a local area network (where strong synchrony characteristics are expected) to a group of computers with unknown processor speeds, communicating on a wide area network (where a large variability in processing and networking delays is expected).

3.1.2.B Synchronous system

In a *synchronous* system, there are known bounds on the time needed to send a message between any pair of processes and on the ratio between the

processing speed of any processes (there is also a bound on the drift rates of the local clocks of each process) [Lyn96].

These bounds allow a large number of distributed problems to be solved. However, this model also has a price: the complexity of handling the variation inherent to a real system is pushed from the algorithms down to the model itself [Fet00, BL91, Jah94, KG94].

The synchronous system requires worst case analyses of execution times of all components in the system. Any modifications to the software or hardware in the system generally require the analysis to be redone, at least partially. Finally, since the bounds on the various delays in a synchronous system must hold even in worst case scenarios, this leads to bounds that are potentially much higher than the most commonly observed values, which can in turn affect the responsiveness of the system.

3.1.2.C Partial synchrony

As seen above, the asynchronous and synchronous system models both have drawbacks: in the former, many fundamental distributed problems are not solvable, whereas the implementation of the latter model is complex. A number of models that address these shortcomings have thus been developed.

Two such models are the *partially synchronous system* [DDS87, DLS88] and the *timed asynchronous model* [CF99]. These models weaken the properties of the synchronous model by either assuming that known bounds exist but only eventually hold forever, or bounds always hold but are unknown. Since these models are weaker than the synchronous model, a real system is more often correctly modeled. The partially synchronous models are however strong enough to allow the solvability of many fundamental distributed problems.

3.1.2.D Oracles

The partially synchronous models *weaken* the synchronous model to account for the asynchrony that exists in a real system. It is also possible to use another approach that allows distributed problems to be solved: the asynchronous model can be *extended* with oracles. These oracles provide additional information to the processes and thus allow previously impossible problems to be solved.

Many different oracles exist. We focus on one specific type of oracle, the failure detector, since it is used by most of the algorithms studied in this thesis. Other oracles are shortly mentioned thereafter.

Unreliable failure detectors With the unreliable failure detector oracle presented in [CT96], each process p_i has a failure detector module FD_i that,

at any time, provides a set of processes it suspects to have crashed. This information may be incorrect (crashed processes might not be suspected and correct processes can be suspected) and may be different for all processes: at a given time t , the failure detector module FD_i of p_i does not necessarily suspect the same set of processes as p_j 's module FD_j .

A failure detector is characterized by two properties: *Completeness* defines which crashed processes are eventually suspected by whom. This property characterizes the failure detector's ability to correctly identify processes that have crashed. The *Accuracy* property restricts the mistakes (*i.e.* suspicions of correct processes) that the failure detector can make.

Together, the completeness and accuracy properties ensure that the failure detector provides useful information. Indeed, if completeness or accuracy were considered alone, then a failure detector suspecting all (*i.e.* all crashed processes), respectively none (*i.e.* no correct process), of the processes would trivially satisfy the considered property. Such a failure detector would however provide no useful information for solving distributed problems.

We now introduce the following properties and define the unreliable failure detectors that are used later on:

Strong completeness Eventually, every process that crashes is permanently suspected by every correct process.

Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct process.

Eventual weak accuracy: There is a time after which some correct process is never suspected by any correct process.

The *Eventually perfect* failure detector $\diamond\mathcal{P}$ is defined by the *Strong completeness* and *Eventual strong accuracy* properties; the *Eventually strong* failure detector $\diamond\mathcal{S}$ is defined by the *Strong completeness* and *Eventual weak accuracy* properties. Both of these failure detectors allow us to solve all the distributed problems that we consider hereafter in the asynchronous system.

Other oracles This paragraph presents some of the other oracles that extend the asynchronous system model and allow consensus and atomic broadcast to be solved.

Coin toss All processes have access to an oracle that outputs random binary values [Ben83]. With such an oracle, consensus can be solved with a probability of 1 [CD89].

Weak ordering oracles A weak ordering oracle allows atomic broadcast to be solved in an asynchronous system by providing processes with

spontaneous ordering of some messages that are broadcast [PSUC02b, PSUC02a]. In local area networks, such a spontaneous ordering naturally occurs.

3.1.3 The Heard-Of model

The Heard-Of model, presented in [CBS06], questions two common assumptions in distributed models: (1) synchrony and failure models are independent characteristics of a distributed model and (2) failures are necessary to analyze the behavior of a distributed algorithm. In the Heard-Of model (noted HO model hereafter), (1) the synchrony and failures are hidden inside a single abstract module and (2) the notion of faulty components is no longer needed. The following paragraphs present the model in further detail.

Algorithms In the HO model, processes communicate in rounds. At the beginning of each round, a process can send a message to the other group members. At the end of the round, each process is then presented with the set of received messages and computes its next state. A message is bound to a round, which implies that if a message m is sent in round r , it can only be received in round r (and is discarded otherwise). If a message is discarded, the HO model does not try to designate a culprit: the message loss could be explained by several factors, including a link failure, a process failure or simply because the message took too long to reach its destination (*i.e.* a synchrony issue).

Each process p has a state $state_p$ (and an initial state $init_p$), a send function S_p^r (that is applied at the beginning of each round r), as well as a state transition function T_p^r . This state transition function maps the state $state_p$ and the set of received messages in round r to a new state. Furthermore, the *heard-of* set $HO(p, r)$ represents the set of processes from which p receives a message in round r . An algorithm \mathcal{A} in the HO model is defined by the $state_p$ and $init_p$ states, and the S_p^r and T_p^r functions.

Figure 3.1 presents an example run of an algorithm in the HO model, from the perspective of process p_1 . At the beginning of round r , p_1 calls its send function $S_{p_1}^r$, which sends messages to p_0 and p_2 . Before the end of round r , p_1 receives a message from p_0 . At the end of round r , p_1 applies the state transition function $T_{p_1}^r$ to its state, with a heard-of set $HO(p_1, r)$ equal to $\{p_0\}$. Round $r + 1$ starts by a call to the $S_{p_1}^{r+1}$ function (which sends a message to p_0). This time, messages are received from p_0 and p_2 . The state transition function $T_{p_1}^{r+1}$ is thus applied with $HO(p_1, r + 1) = \{p_0, p_2\}$.

Predicates The execution of an algorithm in the HO model depends heavily on the $HO(p, r)$ set of processes that a process p hears of during round r .

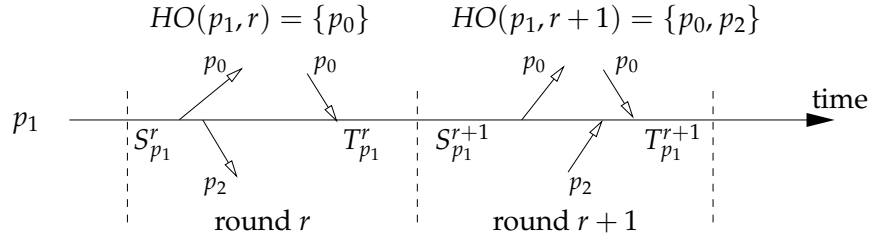


Figure 3.1: Example of two rounds r and $r + 1$ in the Heard-Of model, from the point of view of a process p_1

In order for the algorithm to solve a problem, the $HO(p, r)$ set must provide a number of properties. Consequently, we define a *communication predicate* \mathcal{P} as a predicate over the collections of *heard-of* sets. For example, the following predicate (noted \mathcal{P}_{sp_unif} , for *space uniform*, in [CBS06]) ensures that all processes hear of the same processes in all rounds:

$$\mathcal{P}_{sp_unif} \equiv \forall r > 0, \forall p, q \in \Pi^2 : HO(p, r) = HO(q, r)$$

The algorithm \mathcal{A} and its predicate \mathcal{P} are tightly coupled, since \mathcal{P} is generally tailored to ensure the correctness of \mathcal{A} . \mathcal{A} and \mathcal{P} together define a *Heard-Of machine* as a pair $M = (\mathcal{A}, \mathcal{P})$.

3.2 Agreement problems

Agreement problems are central to distributed systems. In this class of problems, a group of processes have to agree on a common decision, which depends on the problem that is being solved (for example, delivering messages in the same order on all processes or deciding whether to commit or abort a distributed transaction).

Agreement problems are also fundamental in the context of group communication. Group communication aims at providing one-to-many and many-to-many communication to a set of processes: the processes are organized in a group and primitives that allow to communicate directly with the group are provided. These primitives have strong semantics (which are typically not available when using point-to-point connections to communicate with the other processes) which include features such as fault tolerance and message ordering.

The agreement problems that are considered in this thesis and that arise in the context of group communication are presented in the following paragraphs.

3.2.1 Consensus

In the consensus problem, a group of processes Π have to agree on a common value based on proposals of the processes [Fis83, CT96]. Consensus is defined by two primitives: *propose* and *decide*. When a process p calls *propose*(v), we say that p *proposes* v . Similarly, whenever p calls *decide*(v), it *decides* v .

As in [CT96], we specify the (uniform) consensus problem by the four following properties:

Termination: Every correct process eventually decides some value.

Uniform integrity: Every process decides at most once.

Uniform agreement: No two processes (correct or not) decide a different value.

Uniform validity: If a process decides v , then v was proposed by some process in Π .

3.2.2 Reliable broadcast

In the reliable broadcast problem, the correct processes among Π need to eventually agree on a common set of delivered messages. Formally, reliable broadcast is defined by two primitives, *rbroadcast* and *rdeliver*, and the three following properties [HT94]:

Validity If a correct process *rbroadcasts* a message m , then it eventually *rdelivers* m .

Agreement If a correct process *rdelivers* m , then all correct processes eventually *rdeliver* m .

Uniform integrity For any message m , every process *rdelivers* m at most once, and only if m was previously *rbroadcast*.

The *Agreement* property of reliable broadcast allows a faulty process to *rdeliver* a message m without ensuring that all correct processes also *rdeliver* m . *Uniform* reliable broadcast addresses this and is defined by the *Validity* and *Uniform integrity* properties of reliable broadcast, and strengthens the *Agreement* property by extending its definition to *all* processes:

Uniform agreement If any process *rdelivers* m , then all correct processes eventually *rdeliver* m .

3.2.3 Atomic broadcast

In the atomic broadcast problem, defined by the primitives *abroadcast* and *adeliver*, processes have to agree on a the order of delivery of a set of messages, *i.e.* all processes have to deliver the same sequence of messages. Formally, we define (uniform) atomic broadcast by the three properties of uniform reliable broadcast (*Validity*, *Uniform agreement* and *Uniform integrity*) and an additional order property [ADGFT00]:

Uniform total order If some process, correct or faulty, *adelivers* m before m' , then every process *adelivers* m' only after it has *adelivered* m .

This *Uniform total order* property of atomic broadcast is useful, for example, to implement state machine replication [Sch93a], where a set of processes visit the same states by taking deterministic actions based on the same input (the *adelivered* messages).

Token based atomic broadcast

In this chapter, we present a novel atomic broadcast algorithm based on token circulation. We start by discussing the advantages of token based algorithms, then describe the new algorithm in the failure detector system model (see Section 3.1.2.D) and study its performance in a simulated environment. Different variants and optimizations of the algorithm are then discussed, and finally we present the algorithm in the Heard-Of system model (see Section 3.1.3).

4.1 Token based atomic broadcast using unreliable failure detectors

4.1.1 Introduction

4.1.1.A Context

Many atomic broadcast algorithms have been published in the last twenty years. These algorithms can be classified according to the mechanism used for message ordering [DSU04]. *Token circulation* is one important ordering mechanism. In these algorithms, a token circulates among the processes, and the token holder has the privilege to order messages that have been broadcast. Additionally, sometimes only the token holder is allowed to broadcast messages. However, the ordering mechanism is not the only key mechanism of an atomic broadcast algorithm. The mechanism used to tolerate failures is another important characteristic of these algorithms. If we consider asynchronous systems with crash failures, the two most widely used mechanisms to tolerate failures in the context of atomic broadcast algorithms are (i) *unreliable failure detectors* [CT96] and (ii) *group membership* [CKV01]. For example, the atomic broadcast algorithm in [CT96] (to-

gether with a consensus algorithm using the failure detector $\diamond S$ [CT96]) falls into the first category; the atomic broadcast algorithm in [BSS91] falls into the second category.

4.1.1.B Group membership mechanism *vs.* failure detector mechanism.

A group membership service provides a consistent membership information to all the members of a group [CKV01]. Its main feature is to *remove* processes that are suspected to have crashed.¹ In contrast, an unreliable failure detector, *e.g.*, $\diamond S$, does not provide consistent information about the failure status of processes. For example, it can tell to process p that r has crashed, while telling at the same time to process q that r is alive.

Both mechanisms can make mistakes, *e.g.*, by incorrectly suspecting correct processes. However, the cost of a wrong failure suspicion is higher when using a group membership service than when using failure detectors. This is because the group membership service removes suspected processes from the group, a costly operation. *This removal is absolutely necessary for the atomic broadcast algorithm that relies on the membership service: the notification of the removal allows the atomic broadcast algorithm to avoid being blocked.* There is no such removal of suspected processes with a failure detector. Moreover, with a group membership service, the removal of a process is usually followed by the addition of another (or the same) process, in order to keep the same replication degree. So, with a group membership service, a wrong suspicion leads to two costly membership operations: *removal* of a process followed by the *addition* of another process.

In an environment where wrong failure suspicions are frequent,² algorithms based on failure detectors thus have advantages over algorithms based on a group membership service. The cost difference has been experimentally evaluated in [USS03] in the context of two specific (not token based) atomic broadcast algorithm.

Atomic broadcast algorithms based on a failure detector have another important advantage over algorithms based on group membership: *they can be used to implement the group membership service!* Indeed, since a (primary partition) group membership service orders views, it seems intuitive to solve group membership using atomic broadcast: this leads to a much simpler protocol stack than implementing atomic broadcast using group membership [MSW03]. However, this is not possible if atomic broadcast relies on group membership.

¹The comment applies to the so-called *primary-partition* membership [CKV01].

²This typically happens if the timeouts used to suspect processes have been set to small values (*i.e.*, in the order of the average message transmission delay), in order to reduce the time needed to detect the crash of processes.

4.1.1.C Why token based algorithms?

According to [WMK94, AMMS⁺95, MS01], token based atomic broadcast algorithms are extremely efficient in terms of throughput, *i.e.*, the number of messages that can be delivered per time unit. The reason is that these algorithms manage to reduce network contention by using the token (1) to avoid the *ack* explosion problem (which happens if each broadcast message generates one acknowledgment per receiving process), and/or (2) to perform flow control (*e.g.*, a process is allowed to broadcast a message only when holding the token). However, none of the token based algorithms use failure detectors: they all rely on a group membership service.³ It is therefore interesting to try to design token based atomic broadcast algorithms that rely on failure detectors, in order to combine the advantage of failure detectors and of token based algorithms: good throughput (without sacrificing latency) in stable environments, but adapted to frequent wrong failure suspicions.

4.1.1.D Contribution of the chapter

The chapter presents the first token based atomic broadcast algorithm that uses unreliable failure detectors instead of group membership. This result is obtained in several steps. We first give a new and more general definition for token based algorithms (Sect. 4.1.2) and introduce a new failure detector, denoted by \mathcal{R} , adapted to token based algorithms (Sect. 4.1.3). The failure detector \mathcal{R} is shown to be strictly weaker than $\diamond\mathcal{P}$, and strictly stronger than $\diamond\mathcal{S}$. Although $\diamond\mathcal{S}$ is strong enough to solve consensus and atomic broadcast, \mathcal{R} has an interesting feature: the failure detector module of a process p_i only needs to give information about the (estimated) state of p_{i-1} . For p_{i-1} , this can be done by sending *I am alive* messages to p_i only, which is extremely cheap compared to failure detectors where each process monitors all other processes. Moreover, in the case of three processes (a frequent case in practice, tolerating one crash), our token based algorithm works with $\diamond\mathcal{S}$.

Section 4.1.4 concentrates on the consensus problem. First we define two classes of token based algorithms: *token-accumulation* algorithms and *token-coordinated* algorithms. We then focus on the token-accumulation approach and give a consensus algorithm based on the failure detector \mathcal{R} .

An algorithm that solves atomic broadcast is presented in Section 4.1.5. The algorithm is inspired from the token based consensus algorithm of Section 4.1.4. Note that a standard solution consists in solving atomic broadcast by reduction to consensus [CT96]. However, this solution is not adequate here, because the resulting algorithm is highly inefficient. Our atomic

³The group membership mechanism does not necessarily appear explicitly in the algorithm, *e.g.*, in [MS01]. It can be implemented in an ad-hoc way.

broadcast algorithm is derived from our consensus algorithm in a more complex manner. Note that we could have presented only the token based atomic broadcast algorithm. However, the detour through the consensus algorithm makes the explanation easier to understand. Section 4.1.6 compares the performance of our new atomic broadcast algorithm with the Chandra-Toueg atomic broadcast algorithm. Related work is presented in Section 4.1.7 and Section 4.1.8 concludes Section 4.1.

4.1.2 System model and definitions

We assume an asynchronous system composed of n processes, detailed in Chapter 3. The k^{th} successor of a process p_i is $p_{(i+k) \bmod n}$, which is noted p_{i+k} for the sake of clarity. Similarly the k^{th} predecessor of p_i is simply denoted by p_{i-k} . Processes can only fail by crashing. A process that never crashes is said to be *correct*, otherwise it is *faulty*. At most f processes are *faulty*. Finally, the system is augmented with unreliable failure detectors [CT96] (see below).

4.1.2.A Agreement problems

The agreement problems considered in this chapter are shortly reminded. The formal specifications of the problems are presented in Section 3.2.

In the consensus problem, the processes have to agree on a common value, based on the initial proposals of the processes. In the atomic broadcast problem, the processes agree on a common total order delivery of a set of messages.

4.1.2.B Token based algorithms

In a traditional token based algorithm, processes are organized in a logical ring and, for token transmission, communicate only with their immediate predecessor and successor (except during changes in the composition of the ring). This definition is too restrictive for failure detector-based algorithms. We define an algorithm to be *token based* if (1) processes are organized in a logical ring, (2) each process p_i has a failure detector module FD_i that provides information only about its immediate predecessor p_{i-1} and (3) each process communicates only with its $f + 1$ predecessors and successors, where f is the number of tolerated failures.

4.1.2.C Failure detectors

We refer below to two failure detectors introduced in [CT96]: $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$. The eventual perfect failure detector $\diamond\mathcal{P}$ is defined by the following properties: (i) *Strong completeness*: Eventually every process that crashes is per-

manently suspected by every correct process, and (ii) *Eventual strong accuracy*: there is a time after which correct processes are not suspected by any correct process. The $\diamond S$ failure detector is defined by (i) *Strong completeness* and (ii) *Eventual weak accuracy*: there is a time after which some correct process is never suspected by any correct process.

4.1.3 Failure detector \mathcal{R}

For token based algorithms we define a new failure detector denoted \mathcal{R} (stands for *Ring*). Given process p_i , the failure detector attached to p_i only gives information about the immediate predecessor p_{i-1} . For every process p_i , \mathcal{R} ensures the following properties:

Completeness If p_{i-1} crashes and p_i is correct, then p_{i-1} is eventually permanently suspected by p_i , and

Accuracy If p_{i-1} and p_i are correct, there is a time t after which p_{i-1} is never suspected by p_i .

The *weaker/stronger* relationship between failure detectors has been defined in [CT96]. We show that (a) $\diamond P$ is strictly stronger than \mathcal{R} (denoted $\diamond P \succ \mathcal{R}$), and (b) \mathcal{R} is strictly stronger than $\diamond S$ if $n \geq f(f+1) + 1$ (denoted $\mathcal{R} \succ \diamond S$).

Lemma 4.1.1. $\diamond P$ is strictly stronger than \mathcal{R} if $f > 1$.

Proof. This result is easy to establish. From the definition it follows directly that $\diamond P$ is stronger or equivalent to \mathcal{R} , denoted by $\diamond P \succeq \mathcal{R}$. Moreover, when p_i is faulty, then \mathcal{R} provides no information about p_{i-1} :⁴ so $\diamond P \not\equiv \mathcal{R}$ ($\diamond P$ not equivalent to \mathcal{R}). With $\diamond P \succeq \mathcal{R}$ we have that $\diamond P \succ \mathcal{R}$. \square

The relationship between \mathcal{R} and $\diamond S$ is more difficult to establish. We first introduce a new failure detector $\diamond S2$ (Sect. 4.1.3.A), then show that $\diamond S2 \succ \diamond S$ (Sect. 4.1.3.B) and $\mathcal{R} \succeq \diamond S2$ if $n \geq f(f+1) + 1$ (Sect. 4.1.3.C). By transitivity, we have $\mathcal{R} \succ \diamond S$ if $n \geq f(f+1) + 1$.

4.1.3.A Failure detector $\diamond S2$

For the purpose of establishing the relation between \mathcal{R} and $\diamond S$ we introduce the failure detector $\diamond S2$ defined as follows:

Strong completeness Eventually every process that crashes is permanently suspected by every correct process and

Eventual “Double” Accuracy There is a time after which *two* correct processes are never suspected by any correct process.

⁴In the special case of $f = 1$, the information about p_{i-1} can be obtained indirectly, *i.e.*, if $f = 1$, the relation between $\diamond P$ and \mathcal{R} is not strict: $\diamond P \succeq \mathcal{R}$.

4.1.3.B $\diamond S2$ strictly stronger than $\diamond S$

$\diamond S$ and $\diamond S2$ differ in the accuracy property only: while $\diamond S$ requires eventually *one* correct process to be no longer suspected by all correct processes, $\diamond S2$ requires the same to hold for *two* correct processes. From the definition, it follows directly that $\diamond S2 \succ \diamond S$.

4.1.3.C \mathcal{R} stronger than $\diamond S2$ if $n \geq f(f + 1) + 1$

We show that \mathcal{R} is stronger than $\diamond S2$ if $n \geq f(f + 1) + 1$ by giving a transformation of \mathcal{R} into the failure detector $\diamond S2$.

Transformation of \mathcal{R} into $\diamond S2$: Each process p_j maintains a set $correct_j$ of processes that p_j believes are correct.

(i) This set is updated as follows. Each time some process p_i changes its mind about p_{i-1} (based on \mathcal{R}), p_i broadcasts (using a FIFO reliable broadcast communication primitive [HT94]) the message $(p_{i-1}, faulty)$, respectively $(p_{i-1}, correct)$. Whenever p_j receives $(p_i, faulty)$, then p_j removes p_i from $correct_j$; whenever p_j receives $(p_i, correct)$, then p_j adds p_i to $correct_j$.

(ia) For process p_i , if $correct_i$ is equal to Π (no suspected process), the output of the transformation (the two non-suspected processes) is p_0 and p_1 . All other processes are suspected.

(ib) For process p_i , if $correct_i$ is not equal to Π (at least one suspected process), the output of the transformation (the two non-suspected processes) is p_k and p_{k+1} such that k is the smallest index satisfying the following conditions: (a) p_{k-1} is not in $correct_i$, and (b) the $f - 1$ immediate successors $p_{k+1}, \dots, p_{k+f-1}$ are in $correct_i$. Apart from p_k and p_{k+1} , all other processes are suspected.

For example, for $n = 7$, $f = 2$, and $correct_i = \{p_0, p_2, p_3, p_5\}$, the non-suspected processes for p_i are p_2 and p_3 . All other processes are suspected. If $correct_i = \{p_0, p_1, p_2, p_3, p_5\}$, the non-suspected processes for p_i are p_0 and p_1 (the predecessor of p_0 is p_6 , not in $correct_i$). All other processes are suspected.

Lemma 4.1.2. Consider a system with $n \geq f(f + 1) + 1$ processes and the failure detector \mathcal{R} . The above transformation guarantees that eventually all correct processes do not suspect the same two correct processes.

Proof. (i) Consider t such that after t all faulty processes have crashed and each correct process p_i has accurate information about its predecessor p_{i-1} . It is easy to see that there is a time $t' > t$ such that after t' all correct processes agree on the same set $correct_i$. Let us denote this set by $correct(t')$.

(ii) The condition $n \geq f(f + 1) + 1$ guarantees that the set $correct(t')$ contains a sequence of f consecutive processes. Consider the following sequence of processes: 1 faulty, f correct, 1 faulty, f correct, etc. If we repeat the pattern f times, we have f faulty processes in a set of $f(f + 1)$

processes. If we add one correct process to the set of $f(f + 1)$ processes, there is necessarily a sequence of $f + 1$ correct processes. With a sequence of $f + 1$ correct processes, there is a sequence of f consecutive processes in $correct(t')$.

(iii) In the case $correct(t') = \Pi$, p_0 and p_1 are trivially correct.

(iv) In the case $correct(t') \neq \Pi$, consider the sequence of $f + 1$ processes p_k, \dots, p_{k+f} . Since there are at most f faulty processes, at least one process p_l in p_k, \dots, p_{k+f} is correct. If $p_l = p_k$, we are done. Otherwise, if p_l is correct, p_{l-1} is correct as well, since the failure detector of p_l is accurate after t' and does not suspect p_{l-1} . By the same argument, if p_{l-1} is correct, p_{l-2} is correct. By repeating the same argument at most $f - 1$ times, we have that p_k is correct.

(v) In the case $correct(t') \neq \Pi$, we prove now that p_{k+1} is correct. Since p_k is correct and p_{k-1} is not in $correct(t')$ (by the selection rule of p_k and p_{k+1}), p_{k-1} is faulty. Thus, there are at most $f - 1$ faulty processes in the sequence of f processes p_{k+1}, \dots, p_{k+f} . In the special case $f = 1$ ($\{p_{k+1}, \dots, p_{k+f-1}\} = \emptyset$), all processes in p_{k+1}, \dots, p_{k+f} are correct. In the case $f > 1$, there is a non-empty sequence $p_{k+1}, \dots, p_{k+f-1}$ in $correct(t')$. Furthermore, there are at most $f - 1$ faulty processes among the f processes p_{k+1}, \dots, p_{k+f} . By the same argument used to show that p_k is correct, we can show that p_{k+1} is correct. \square

The transformation of \mathcal{R} into $\diamond S2$ ensures the *Eventual "double" accuracy* property if $n \geq f(f + 1) + 1$. Since all processes except two correct processes are suspected, the *Strong completeness* property also holds. Consequently, if $n \geq f(f + 1) + 1$ we have $\mathcal{R} \succeq \diamond S2$.

4.1.4 Token based consensus

4.1.4.A Two classes of algorithms

We identify two classes of token based consensus algorithms: *token-accumulation* algorithms and *token-coordinated* algorithms. In the *token-accumulation* algorithms, each token holder votes for the proposal transported in the token. Votes are accumulated as the token circulates and once enough votes have been collected, the token holder can decide. In this class of algorithms, the only communication is related to the circulation of the token. This is not the case of *token-coordinated* algorithms. In these algorithms the token holds a proposal, but, in order to decide, the token holder can communicate with all other processes. Algorithms based on the *rotating-coordinator paradigm* (such as the Chandra-Toueg $\diamond S$ consensus algorithm [CT96]) can easily be adapted to this class ([MK00] describes such a transformation). Token-accumulation algorithms are more genuine token based algorithms,

and the chapter concentrates on this class of algorithms. Henceforth, *token-accumulation* algorithms will simply be referred to as *token based* algorithms.

4.1.4.B Token circulation

The token circulation is as follows. To avoid the loss of the token due to crashes, process p_i sends the token to its $f + 1$ successors in the ring, *i.e.*, to $p_{i+1}, \dots, p_{i+f+1}$.⁵ Furthermore, when awaiting the token, process p_i waits to get the token from p_{i-1} , unless it suspects p_{i-1} . If p_i suspects p_{i-1} , it accepts the token from any of its predecessors (see Algorithm 4.1).

Algorithm 4.1: Receive token (code of process p_i)

- 1: **wait until** received token from p_{i-1} **or** $p_{i-1} \in \mathcal{D}_{p_i}$ {query failure detector \mathcal{D}_{p_i} }
 - 2: **if** token not received **then** {accept from anyone}
 - 3: **wait until** received token from $p \in \{p_{i-f-1}, \dots, p_{i-1}\}$
-

4.1.4.C Token based consensus algorithm

Basic idea Consensus is achieved by passing a token between the different processes. The token contains information regarding the current proposal (or the decision once it has been taken). The token is passed between the processes on a logical ring p_0, p_1, \dots, p_{n-1} . Each token holder “votes” for the proposal in the token and then sends it to its neighbors. As soon as a sufficient number of token holders have voted for some proposal v , then v is decided. The decision is then propagated as the token circulates along the ring.

Naive algorithm We start by presenting a naive algorithm that illustrates both the basic idea behind our algorithm and its difficulty. Let the token carry an *estimate* value (denoted by *token.estimate*) and the number of votes for this estimate (denoted *token.votes*). Let each process p_i , upon receiving the token, blindly adopt the token proposal (which is stored in *estimate_i* and add its vote to the proposal (see Algorithm 4.2). Obviously, this naive algorithm does not work: it would solve consensus in an asynchronous system, in contradiction with the FLP impossibility result [FLP85].

Overview of the token based consensus algorithm As just shown, a token based algorithm cannot blindly increase the votes accumulated. We slightly change the above behavior. The processes need one additional

⁵The token should be seen as a *logical* token. Multiple backup copies circulate in the ring, but they are discarded by the algorithm if no suspicion occurs. Henceforth, the *logical* token will simply be referred to as “the token”.

4.1. Token and failure detector based atomic broadcast

Algorithm 4.2: Token handling by p_i (option 1)

```

estimatei ← token.estimate
token.votes ← token.votes + 1
if token.votes ≥  $f + 1$  then
    decide(token.estimate)
    send token to  $\{p_{i+1}, \dots, p_{i+f+1}\}$ 

```

information: the *gap* in the circulation of the token. When a process p_i receives the token from process $sender \equiv p_j$, the *gap* is $i - j - 1$, denoted by $gap(sender \rightarrow p_i)$. We have $gap(sender \rightarrow p_i) = 0$ only if the token is received from the immediate predecessor. Upon receiving the token, a process does the following (see Algorithm 4.3):

As long as there is no gap in the token circulation *token.votes* is incremented by the receiver p_i . If at that point *token.votes* is greater than the vote threshold $f + 1$, p_i decides on the estimate of the token. The decision is then propagated with the token.

Algorithm 4.3: Token handling by p_i (option 2)

```

if ( $gap(sender \rightarrow p_i) \neq 0$ ) then
    token.votes ← 0 {reset token}
    estimatei ← token.estimate
    token.votes ← token.votes + 1
    if token.votes ≥  $f + 1$  then
        decide(token.estimate)
    send token to  $\{p_{i+1}, \dots, p_{i+f+1}\}$ 

```

Conditions for Agreement vs. Termination In the above algorithm, where votes are reset as soon as a gap in the token circulation is detected, *Agreement* holds if the vote threshold is greater or equal than $f + 1$. *Termination* additionally requires the failure detector \mathcal{R} and that there be at least $n \geq (f + 1)f + 1$ processes in the system.

Remark: The condition $gap(sender \rightarrow p_i) = 0$ is not a necessary condition for *Agreement* in a token based consensus algorithm. In [ES03], we present an algorithm without this condition and which is instead parameterized with *gapThreshold* (the number of gaps in the token circulation before resetting the vote counter) and *voteThreshold* (the number of votes required to decide). *Agreement* holds if $voteThreshold \geq (gapThreshold + 1)f + 1$. However *Termination* still requires $gapThreshold = 0$ (in addition to $n \geq (f + 1)f + 1$ and \mathcal{R}).

The algorithm that allows $gapThreshold > 0$ is not further presented here since (1) it is more complex than the algorithm that does not allow

gaps in the token circulation, (2) it anyhow requires $gapThreshold = 0$ for *Termination* to hold and (3) the generalized algorithm does not improve the message complexity or time complexity of the algorithm that requires $gap(sender \rightarrow p_i) = 0$.

Details of the algorithm The token contains the following fields: *round* (round number), *estimate*, *votes* (accumulated votes for the *estimate* value) and *decision* (a boolean indicating if *estimate* is the decision).

Algorithm 4.4: Token-accumulation consensus (code of p_i)

```

1: upon propose( $v_i$ ) do
2:    $estimate_i \leftarrow v_i$ ;  $decided_i \leftarrow false$ ;  $round_i \leftarrow 0$ 
3:   if  $p_i = p_0$  then                                     {send token with (round, estimate, votes, decision flag)}
4:      $send(0, v_0, 1, false)$  to  $\{p_1, \dots, p_{f+1}\}$ 
5:   else if  $p_i \in \{p_{n-f}, \dots, p_{n-1}\}$  then           {send "dummy" token}
6:      $send(-1, v_i, 0, false)$  to  $\{p_1, \dots, p_{i+f+1}\}$ 

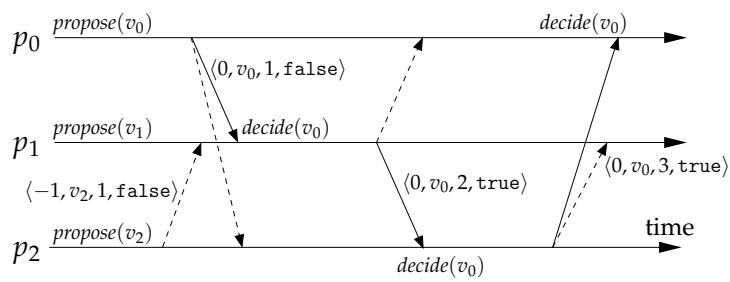
7: Token handling by  $p_i$ :
8:   loop
9:      $token \leftarrow receive\_token(round_i)$                  {see Proc. 4.1}
10:    if not  $decided_i$  then
11:       $estimate_i \leftarrow token.estimate$ 
12:      if ( $gap(sender \rightarrow p_i) = 0$ ) then
13:         $votes_i \leftarrow token.votes + 1$                  {add vote}
14:      else
15:         $votes_i \leftarrow 1$ ;                               {reset votes}
16:      if ( $votes_i \geq f + 1$ ) or  $token.decision$  then
17:         $decide(estimate_i)$ ;  $decided_i \leftarrow true$ 
18:       $token \leftarrow (round_i, estimate_i, votes_i, decided_i)$ 
19:       $send\ token\ to\ \{p_{i+1}, \dots, p_{i+f+1}\}$ 
20:       $round_i \leftarrow round_i + 1$ 
21:    upon reception of token s.t.  $token.round < round_i$  do
22:      if  $token.decision$  and (not  $decided_i$ ) then
23:         $estimate_i \leftarrow token.estimate$ 
24:         $decide(estimate_i)$ ;  $decided_i \leftarrow true$ 

```

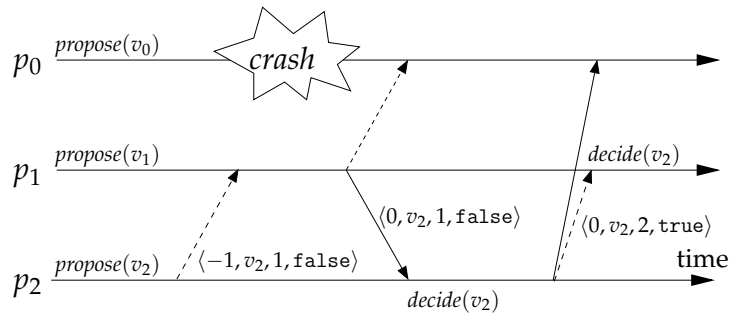
The initialization code is given by lines 1 to 6 in Algorithm 4.4. Lines 5-6 show the *dummy* token sent to prevent blocking in the case processes p_0, \dots, p_{f-1} are initially crashed. A dummy token has $round = -1$ and $votes = 0$, and is sent only to processes $\{p_1, \dots, p_f\}$. The *estimate* of this token is the proposed value v_i of the sender process p_i .

The token handling code is given by lines 7 to 24 in Algorithm 4.4. At

4.1. Token and failure detector based atomic broadcast



(a) No crash, no suspicion



(b) p_0 crashes

Figure 4.1: Example execution of the consensus algorithm

line 9, process p_i starts by receiving the token (see Algorithm 4.1) for the expected $round_i$.⁶ If p_i has not yet decided, then p_i starts by updating its estimate (line 11). If there was no gap in the token circulation, then the votes are incremented (line 13). Otherwise, the votes are reset to 1 (line 15), which starts a new sequence of vote accumulation. At line 16, process p_i checks whether there are enough votes for a decision to be taken. If so, p_i decides (line 17). Finally, the token with the updated fields is sent to the $f + 1$ successors (line 19), and process p_i increments $round_i$ (line 20).

Lines 8-20 ensure that at least one correct process eventually decides. However, if $f > 1$, this does not ensure that all correct processes eventually decide. Consider the following example: p_i is the first process to decide, p_{i+1} is faulty. In this case, p_{i+2} may always receive the token from p_{i-1} , a token that does not carry a decision; p_i might be the only process to ever decide. Lines 21-24 ensure that every correct process eventually decides. The token received at line 8, for $round_i$, follows Algorithm 4.1. Other tokens are received at line 21: if the token carries a decision, process p_i decides. Note that the stopping of the algorithm is not discussed here. It can easily be added.

Figure 4.1 presents an example execution of the consensus algorithm in a system with $n = 3$ processes. The dashed arrows correspond to “backup” tokens (that are used only when failures or suspicions occur) whereas the solid arrows show the main token (transmitted between process p_i and p_{i+1}). When no crashes nor suspicions occur (Figure 4.1(a)), process p_1 receives p_0 's token and increments the votes for p_0 's proposal v_0 . With two votes, p_1 decides v_0 . The token is then passed on to p_2 (with the *decided* flag set to `true`) who decides. Finally, p_0 decides after the last token transmission.

In the case of a crash of p_0 , p_1 eventually suspects p_0 and thus accepts p_2 's token (with p_2 's proposal v_2). Since there is a gap in the token circulation, the votes are reset (no decision can be taken) and the token is sent from p_1 to p_2 . Process p_2 receives the token, increments the votes and decides v_2 . Process p_1 then decides one communication step later after receiving the token (with the decision flag).

Proof of the token based algorithm The proofs of the *Uniform validity* and *Uniform integrity* properties are easy and omitted. A proof of the *Uniform agreement* and *Termination* properties of the token-accumulation consensus algorithm are presented in the following paragraphs.

Uniform agreement. Let p_i be the first process to decide (say at time t), and let v be the decision value. By line 16 of Algorithm 4.4, we have $votes_i \geq$

⁶To avoid complicated notation, we implicitly assume that, for process p_i , waiting a token for $round_i$ means either (1) waiting a token from p_j , $j < i$, with $token.round = round_i$, or (2) waiting a token from p_j , $j > i$, with $token.round = round_i - 1$.

$f + 1$. Votes are reset for each gap. So, $votes_i \geq f + 1$ ensures that at time t , all processes $p_j \in \{p_{i-1}, \dots, p_{i-f}\}$, have $estimate_j = v$. Any process p_k , successor of p_i in the ring, receives the token from one of the processes p_i, \dots, p_{i-f} . Since all these processes have their estimate equal to v , the token received by p_k necessarily carries the estimate v . So after t , the only value carried by the token is v , *i.e.*, any process that decides will decide v . \square

Termination. Assume at most f faulty processes and the failure detector \mathcal{R} . We show that, if $n \geq f(f + 1) + 1$, then every correct process eventually decides.

First it is easy to see that the token circulation never stops: if p_i is a correct process that does not have the token at time t , then there exists some time $t' > t$ such that p_i receives the token at time t' . This follows from (1) the fact that the token is sent by a process to its $f + 1$ successors, (2) the *receive token* procedure (Algorithm 4.1), and (3) the completeness property of \mathcal{R} (which ensures that if p_i waits for the token from p_{i-1} and p_{i-1} has crashed, then p_i eventually suspects p_{i-1} and accepts the token from any of its $f + 1$ predecessors).

The second step is to show that at least one correct process eventually decides. Assume the failure detector \mathcal{R} , and let t be such that after t no correct process p_i is suspected by its immediate correct successor p_{i+1} . Since we have $n \geq f(f + 1) + 1$ there is a sequence of $f + 1$ correct processes in the ring. Let p_i, \dots, p_{i+f} be this sequence. After t , processes p_{i+1}, \dots, p_{i+f} only accept the token from their immediate predecessor. Thus, after t , the token sent by p_i is received by p_{i+1} , the token sent by p_{i+1} is received by p_{i+2} , and so forth until the token sent by p_{i+f-1} is received by p_{i+f} . Once p_{i+f} has executed line 13 of Algorithm 4.4, we have $votes_i \geq f + 1$. Consequently, p_{i+f} decides.

Finally, if one correct process p_k decides, and sends the token with the decision to its $f + 1$ successors, the first correct successor of p_k , by line 21 or line 9, eventually receives the token with the decision and decides (if it has not yet done so). By a simple induction, every correct process eventually also decides. \square

4.1.5 Token based atomic broadcast algorithms

In this section we show how to transform the token based consensus algorithm into an atomic broadcast algorithm. Note that we could have presented the atomic broadcast algorithm directly. However, since the consensus algorithm is simpler than the atomic broadcast algorithm, we believe that a two-step presentation makes it easier to understand the atomic broadcast algorithm.

Note also that it is well known how to solve atomic broadcast by reduction to consensus [CT96]. However, the reduction, which transforms atomic broadcast into a sequence of consensus, yields an inefficient algorithm here. The reduction would lead to multiple instances of consensus, with one token per consensus instance. We want a single token to “glue” the various instances of consensus.

To be correct, the atomic broadcast algorithm requires the failure detector \mathcal{R} , a number of processes $n \geq f(f + 1) + 1$, and a vote threshold at $f + 1$ in order to decide, as was the case in the consensus algorithm above.

4.1.5.A Overview

In the token based atomic broadcast algorithm, the token transports (i) sets of messages and (ii) sequences of messages. More precisely, the token carries the following information: (*round*, *proposalSeq*, *votes*, *adeliv*, *nextSet*). Messages in the sequence *proposalSeq* are delivered as soon as a sufficient number of consecutive “votes” have been collected. The field *adeliv* is the set of all consensus decisions that the token is aware of (*i.e.*, a set of pairs associating a consensus number to a sequence of messages). When a process receives the token, it can therefore, if needed, catch up with the message deliveries performed by other processes.

Finally, while the token accumulates votes for *proposalSeq*, it simultaneously collects in *nextSet* the messages that have been *abroadcast*, but not *adelivered* yet. The set *nextSet* grows as the token circulates. Whenever messages in *proposalSeq* can be delivered, *nextSet* is used as the proposal for the next decision.

4.1.5.B Details

Each process p_i manages the following data structures (see Algorithm 4.5): *round_i* (the current round number), *abroadcast_i* (the set of all messages that have been *abroadcast* by p_i or another process, and not yet ordered), *adeliv_i* (the set of all ordered messages that p_i knows of, represented as a set of pairs associating a consensus number to a sequence of messages) and *nextCons_i* (the sequence number of the next consensus execution). For reasons of space, the algorithm is split into two parts.

Algorithm 4.5 presents the initialization of the atomic broadcast algorithm, as well as the *abroadcast* and *adelivery* of messages: *delivery(seq)* is called by Algorithm 4.6.

Algorithm 4.6 describes the token-handling. Lines 4 to 14 of Algorithm 4.6 correspond to lines 9 to 17 of the consensus algorithm (presented in Algorithm 4.4). The procedure *delivery()* is called to *adeliver* messages (line 13). When this happens, a new sequence of messages can be proposed for delivery. This is done at lines 15 to 17. Finally, lines 21-24 handle the

4.1. Token and failure detector based atomic broadcast

Algorithm 4.5: Token-accumulation atomic broadcast – part 1 (code of p_i)

```

1: Initialisation:
2:    $abroadcast_i \leftarrow \emptyset; adeliiv_i \leftarrow \emptyset; round_i \leftarrow 0$ 
3:    $nextCons_i \leftarrow 1$ 
4:   if  $p_i = p_0$  then           {send token with (round, proposalSeq, votes, adeliiv, nextSet)}
5:      $send(0, abroadcast_0, 1, \emptyset, abroadcast_0)$  to  $\{p_1, \dots, p_{f+1}\}$ 
6:   else if  $p_i \in \{p_{n-f}, \dots, p_{n-1}\}$  then           {send "dummy" token}
7:      $send(-1, \emptyset, 0, \emptyset, \emptyset)$  to  $\{p_1, \dots, p_{i+f+1}\}$ 

8: abroadcast and adeliiv
9:   To execute  $abroadcast(m)$  :
10:     $abroadcast_i \leftarrow abroadcast_i \cup \{m\}$ 
11:   To execute  $deliiv(seq)$ :
12:    while  $\exists (nextCons_i, msgs) \in seq$  do
13:       $adeliiv_i \leftarrow adeliiv_i \cup \{(nextCons_i, msgs)\}$ 
14:       $abroadcast_i \leftarrow abroadcast_i \setminus msgs$ 
15:      adeliiv messages in  $msgs$ 
16:       $nextCons_i \leftarrow nextCons_i + 1$ 

```

reception of other tokens. This is needed for *Uniform agreement* and *Validity* when $f > 1$. Lines 22 and 23 are for *Uniform agreement* (they play the same role as lines 22-24 of Algorithm 4.4). Line 24 is for *Validity* (consider $f = 2$, p_i correct and p_{i+1} faulty; without line 24, process p_{i+2} might, in all rounds, receive the token only from p_{i-1} ; if this happens, messages *abroadcast* by p_i would never be *adeliiv*ed).

The proof of the algorithm can be derived from the proof of the token based consensus algorithm.

4.1.5.C Optimizations

The following paragraphs present optimizations that can be applied to the token based atomic broadcast algorithm presented in Algorithms 4.5 and 4.6. Later, in Section 4.2, we further discuss different variants of the algorithm that all ensure the *Uniform agreement* property of atomic broadcast, but with different requirements on memory and token transmissions.

In our algorithm, the token carries whole messages, rather than only message identifiers. This solution is certainly inefficient. The algorithm can be optimized so that only the message identifiers are included in the token. This can be addressed by adapting techniques presented in other token based atomic broadcast algorithms, *e.g.*, [CM84, MS01], and is thus not discussed further.

The optimization above reduces the size of the token but does not pre-

Algorithm 4.6: Atomic broadcast: token handling by p_i – part 2

```

1: loop
2:   token  $\leftarrow$  receive-token( $round_i$ ) {see Algorithm 4.1}
3:    $abroadcast_i \leftarrow abroadcast_i \cup token.proposalSeq \cup token.nextSet \setminus$ 
      $\{msgs | (-, msgs) \in adeli_v_i\}$ 
4:   if  $|token.adeli_v| < |adeli_v_i|$  then {“old” token}
5:     token.proposalSeq  $\leftarrow \emptyset$ 
6:   else {token with new information}
7:     delivery(token.adeli_v)
8:     if (token received from  $p_{i-1}$ ) and ( $token.proposalSeq \neq \emptyset$ ) then
9:       token.votes  $\leftarrow$  token.votes + 1
10:    else
11:      token.votes  $\leftarrow$  1
12:    if ( $token.votes \geq f + 1$ ) then
13:      delivery( $\{(nextCons_i, token.proposalSeq)\}$ )
14:      token.proposalSeq  $\leftarrow \emptyset$ 
15:    if  $token.proposalSeq = \emptyset$  then {new proposal}
16:      token.proposalSeq  $\leftarrow abroadcast_i$ 
17:      token.votes = 1
18:    token  $\leftarrow (round_i, token.proposalSeq, token.votes, adeli_v_i, abroadcast_i)$ 
19:    send token to  $\{p_{i+1}, \dots, p_{i+f+1}\}$ 
20:     $round_i \leftarrow round_i + 1$ 
21: upon reception of token s.t.  $token.round < round_i$  do
22:   if  $|token.adeli_v| > |adeli_v_i|$  then
23:     delivery(token.adeli_v)
24:    $abroadcast_i \leftarrow abroadcast_i \cup token.nextSet$ 

```

vent it from growing indefinitely. This can be handled as follows. Consider a process p that receives the token with the sequence s_1 in the field *adeli*v and later, in a different round, receives the token with a longer sequence s_2 in the same field (s_1 is a subsequence of s_2). When p receives the token with the sequence s_2 , the token containing sequence s_1 has been received by at least $f + 1$ processes, *i.e.*, by at least one correct process. The sequence s_1 can thus be removed from the token. In nice runs (no failures, no suspicions), this means that a process that delivers new messages in round i (thus increasing the size of the *adeli*v sequence in the token) then removes those messages from the token in round $i + 1$.

The circulation of the token can also be optimized. If all processes are correct, each process actually only needs to send the token to its immediate successor. So, by default each process p_i only sends the token to p_{i+1} . This approach requires that if process p_i suspects its predecessor p_{i-1} , it must send a message to its predecessors p_{i-f} to p_{i-2} , requesting the token.⁷ A process, upon receiving such a message, sends the token to p_i . If all processes are correct, this optimization requires only a single copy of the token to be sent by each token-holder instead of $f + 1$ copies, thus reducing the network contention due to the token circulation by a factor $f + 1$.

Finally, with Algorithm 4.6, a single proposal is contained in the token. Since a consensus decision is taken after f communication steps, this means that as f increases, consensus decisions are taken at an increasingly slower rate. To achieve higher throughputs, it is thus essential to be able to fit several proposals in a single token. This can be done by associating votes separately to several proposals (instead of a single proposal with a single vote in Algorithm 4.6). Moreover, each proposal is associated with the number of the consensus execution it has been proposed in, which is used as a tie-breaker in case several proposals reach $f + 1$ votes at the same time. This optimization does however have one drawback on the performance of the algorithm: the additional proposals increase the size of the token. This implies that even with this optimization, the throughput of the algorithm in the case of $f > 1$ remains lower than in the case of $f = 1$.

4.1.6 Simulation Results

In this section we compare the performance of our new atomic broadcast algorithm with the Chandra-Toueg algorithm, in which atomic broadcast is solved by reduction to consensus [CT96]. The Chandra-Toueg algorithm does not use failure detectors directly, but relies solely on consensus (which in turn relies on failure detectors).⁸ For consensus, we consider two differ-

⁷This request should only be sent once during each round, to avoid an explosion of request messages in the case of very frequent wrong suspicions

⁸This allows us to compare two different atomic broadcast algorithms, both using failure detectors (directly, as in the token based algorithm, or indirectly, as in the reduction to

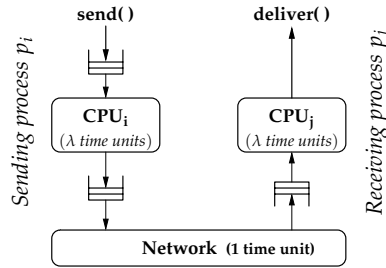


Figure 4.2: The Neko simulation model

ent algorithms: (1) the Chandra-Toueg consensus algorithm (*CT*), based on a centralized communication schema [CT96], and (2) the Mostéfaoui-Raynal consensus algorithm (*MR*), based on a decentralized communication schema [MR99]. The two algorithms use the failure detector $\diamond\mathcal{S}$ and require $f < n/2$. The details of the atomic broadcast and consensus algorithms are presented in Appendix A. The implementation of our new token based algorithm (noted *TokenFD*) is optimized in that messages that are *abroadcast* are initially sent to all processes (so that they are added to a token faster) and decisions are broadcast as soon as they are taken (so that the decision reaches all processes faster, allowing messages to be *adelivered* faster). The comparison is done by simulation.

4.1.6.A Simulation model and parameters

The results have been obtained using the Neko simulation and prototyping framework [UDS02]. Using this framework, the same (Java) implementation of a protocol can be used in a simulated environment and on a real network. The message transmission has been modeled as in [USS03] and [UDS00] and is illustrated in Figure 4.2.

Both the network and the hosts can be a bottleneck. Each CPU (for sending and receiving messages) and the network are modeled as resources that need to be acquired, used, and finally released. A message m transmitted from process p_i to process p_j (i) first uses the CPU of p_i (with a cost of λ), (ii) then the network (with a cost of 1), and (iii) finally the CPU of p_j (with a cost of λ), as shown in Figure 4.2. If a resource (the CPU or the network) is already in use, message m is enqueued until the resource is free. The parameter λ ($\lambda \geq 0$) models the relative speed of processing a message on a host compared to transmitting it over the network: $\lambda = 1$ indicates that CPU processing and transmitting over the network have the same cost, $\lambda > 1$ indicate that CPU processing is expensive compared to transmitting

consensus algorithm, where consensus uses failure detectors).

over the network, $\lambda < 1$ indicates that transmitting over the network is expensive compared to CPU processing. We use three representative values $\{0.1, 1, 10\}$ for λ and simulate the algorithms on a multicast network.

4.1.6.B Performance Metric : Latency versus Throughput

We evaluate the performance of the algorithms with four types of faultloads, as in [USS03]: *normal-steady* (no failures, no suspicions), *crash-steady* (one or two failures occur before the start of the run, no wrong suspicions), *crash-transient* (failures are injected during the run and detected after a detection time T_D , the performance is measured during the period of instability that follows a crash) and *suspicion-steady* (no failures, but wrong suspicions). In the *suspicion-steady* faultload, wrong suspicions occur on average every T_{MR} time units (the mistake recurrence time) and last on average T_M time units (the average duration). When T_{MR} is low, wrong suspicions occur frequently, whereas when T_M is high, wrong suspicions take a long time to be corrected.

All of these tests were run with two system settings: (1) $f = 1$: one tolerated failure ($n = 3$ processes for *CT*, *MR* and *TokenFD*) and (2) $f = 2$: two tolerated failures ($n = 5$ processes for *CT* and *MR*, compared to $n = 7$ processes for *TokenFD*).⁹

We use a simple symmetric workload: all processes send atomic broadcasts at the same rate, and the overall rate is called *throughput*. The performance metric for the algorithms is the *latency*, defined as the *average* (over all correct processes) of the elapsed time between sending a message m and the delivery of m .

The results are shown in Figures 4.3 to 4.14. The graphs give the *latency* as a function of the overall throughput or the mistake recurrence time (in the *suspicion-steady* faultload). We set the time unit of the network simulation model to 1 ms, to make sure that the reader is not distracted by an unfamiliar presentation of time/frequency values (one that refers to time units). Any other value could have been used. The 95% confidence interval is shown for each point in the graphs.

4.1.6.C One tolerated failure ($f = 1$)

In the case $f = 1$, all algorithms need a system with $n = 3$ processes to guarantee liveness. In such a setting, and with a *normal-steady* faultload (*i.e.* no failures, no wrong suspicions), the *TokenFD* algorithm needs two

⁹ The number of processes might seem small, but is adequate to implement scalable atomic broadcast algorithms. Indeed, in a system with a large amount of processes, there is typically a small kernel of “servers” that order the messages and then broadcast them to all other processes. This issue is further discussed in Chapter 9.

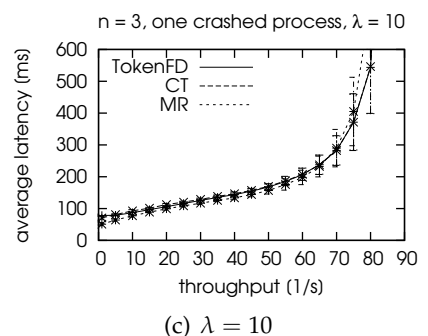
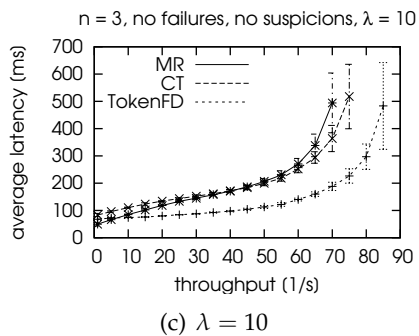
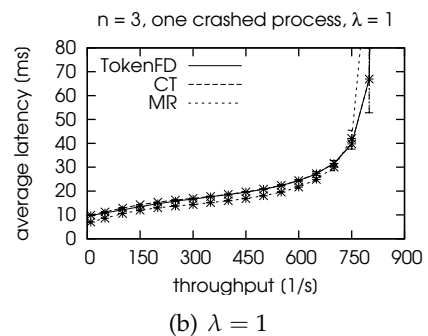
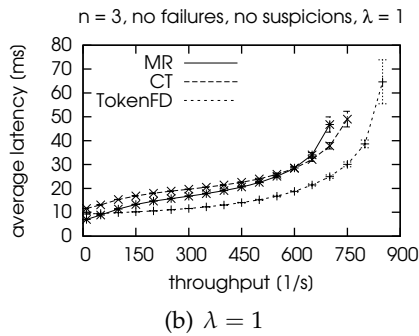
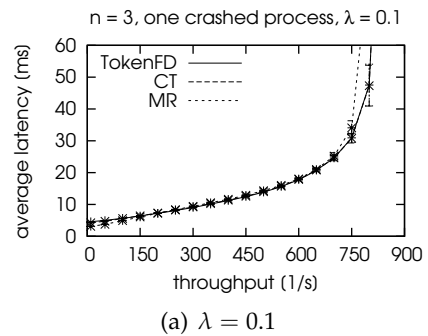
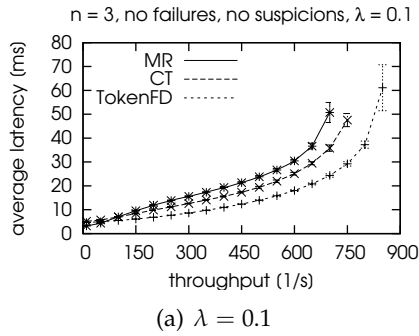
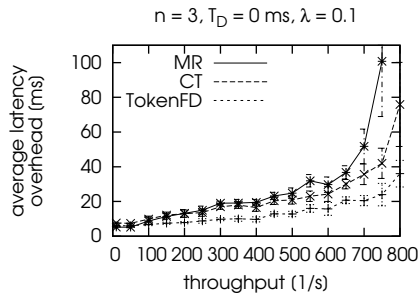


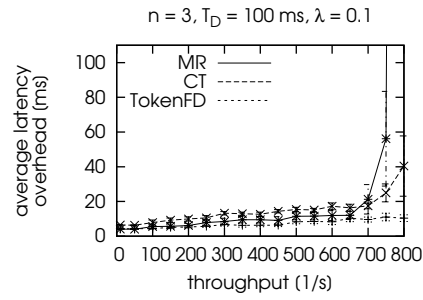
Figure 4.3: Latency vs. throughput with a *normal-steady* faultload, $n = 3$ correct processes

Figure 4.4: Latency vs. throughput with a *crash-steady* faultload, one crashed process ($n = 3$ processes)

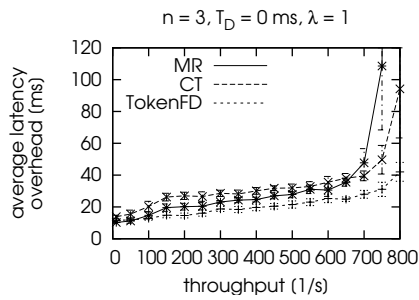
4.1. Token and failure detector based atomic broadcast



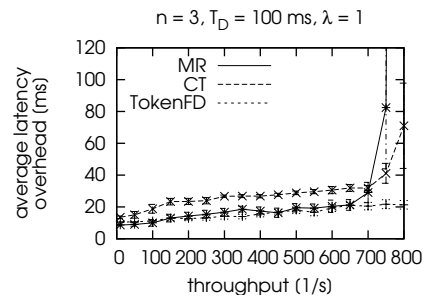
(a) $\lambda = 0.1$



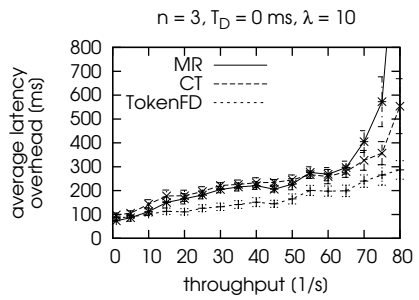
(a) $\lambda = 0.1$



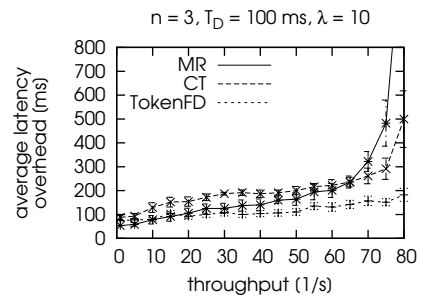
(b) $\lambda = 1$



(b) $\lambda = 1$



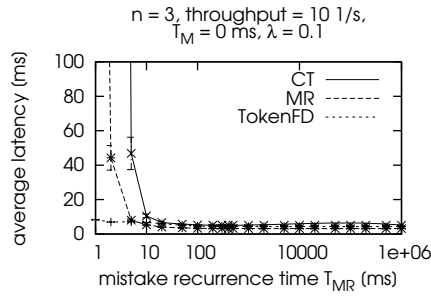
(c) $\lambda = 10$



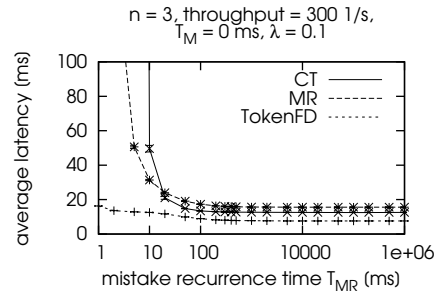
(c) $\lambda = 10$

Figure 4.5: Latency overhead vs. throughput with a *crash-transient* faultload, one crash (in a group of $n = 3$ processes), detection time $T_D = 0\text{ms}$

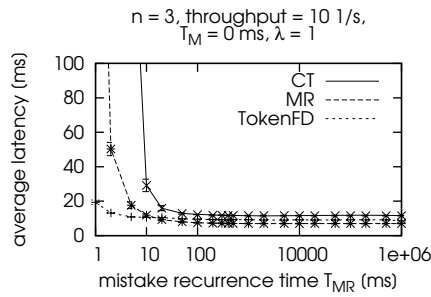
Figure 4.6: Latency overhead vs. throughput with a *crash-transient* faultload, one crash (in a group of $n = 3$ processes), detection time $T_D = 100\text{ms}$



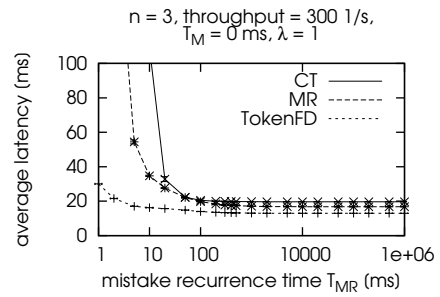
(a) $\lambda = 0.1$



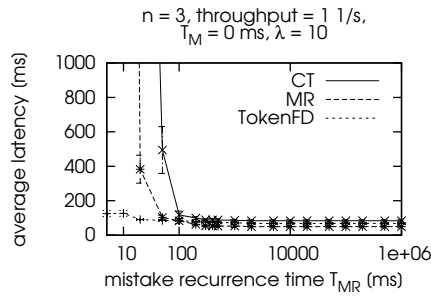
(a) $\lambda = 0.1$



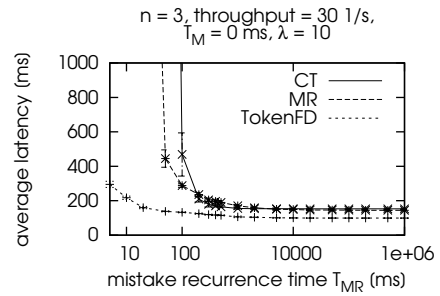
(b) $\lambda = 1$



(b) $\lambda = 1$



(c) $\lambda = 10$



(c) $\lambda = 10$

Figure 4.7: Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload, $n = 3$ processes, mistake duration $T_M = 0ms$, throughput of 10 ($\lambda = 0.1, 1$) or 1 ($\lambda = 10$) *abroadcasts* per second

Figure 4.8: Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload, $n = 3$ processes, mistake duration $T_M = 0ms$, throughput of 300 ($\lambda = 0.1, 1$) or 30 ($\lambda = 10$) *abroadcasts* per second

broadcast messages and one point-to-point message (*i.e.* three communication steps) for all processes to *abroadcast* and *adeliver* a message. The *CT* (atomic broadcast and consensus) algorithm needs $n = 3$ point-to-point messages and 3 broadcast messages, for a total of 4 communication steps. Finally, the *MR* algorithm (coupled with Chandra-Toueg atomic broadcast) needs $2n + 1 = 7$ broadcasts, for a total of 3 communication steps. According to this complexity analysis, the *TokenFD* algorithm should perform better than the *CT* and *MR* algorithms in a system with 3 processes. Figure 4.3 confirms this analysis in the case of a run without failures: the *TokenFD* algorithm achieves lower latencies than both other algorithms for all loads but the lowest.

In the case of one faulty process (*crash-steady* faultload), the performance gap between the *TokenFD* algorithm and both other algorithms is negligible (Figure 4.4), due to two factors: (1) the decrease of the network contention (only two processes try to access the network) which is favorable to the *CT* and *MR* algorithms and (2) the reduced number of decisions that the *TokenFD* atomic broadcast algorithm can take when a process crashes: if p_i crashes, then the votes in the token sent from p_{i-1} to p_{i+1} are always reset. If no crashes (and no suspicions occur), the *TokenFD* algorithm never needs to reset votes. In the scenario of one crash in a system with three processes, the communication patterns of the *CT*, *MR* and *TokenFD* algorithms are almost identical, which explains the similar performance.

In runs with a *crash-transient* faultload, if the detection is very fast (modeled as detection time $T_D = 0$), the *TokenFD* algorithm performs better than both other algorithms, as is shown in Figure 4.5. When the detection time is $T_D = 100ms$, the *TokenFD* algorithm still achieves a slightly lower latency overhead than both other algorithms (Figure 4.6).

Finally, in runs with wrong suspicions (*suspicion-steady* faultload), the *TokenFD* algorithm achieves lower latencies than the other algorithms, especially when failure detector mistakes are frequent (small values of T_{MR}). Figures 4.7 and 4.8 illustrate this for low and high throughputs respectively.

4.1.6.D Two tolerated failures ($f = 2$)

In the case of $f = 2$, *CT* and *MR* need a system with $n = 5$ processes, whereas the *TokenFD* algorithm needs $n = 7$ processes to guarantee liveness. In such a setting, and with a *normal-steady* faultload (*i.e.* no wrong suspicions), the *TokenFD* algorithm needs two broadcast messages and 2 point-to-point messages (*i.e.* four communication steps). The results for the *CT* and *MR* consensus algorithms are as before: $n = 5$ point-to-point messages and 3 broadcasts for a total of 4 communication steps for *CT*, $2n + 1 = 11$ broadcasts for a total of 3 communication steps for *MR*.

So, roughly speaking, the *TokenFD* algorithm appears better in terms

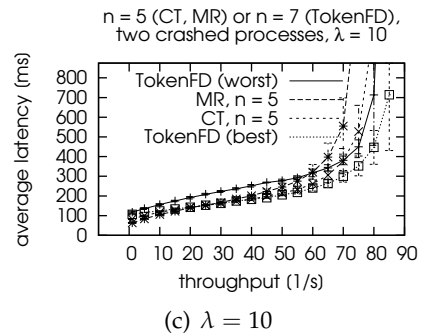
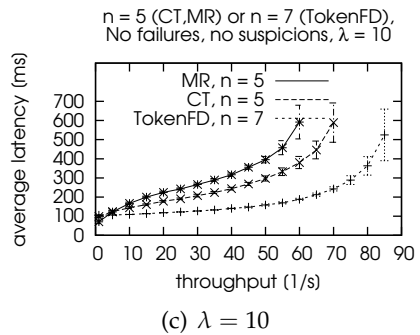
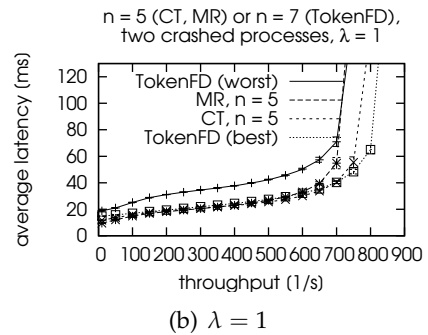
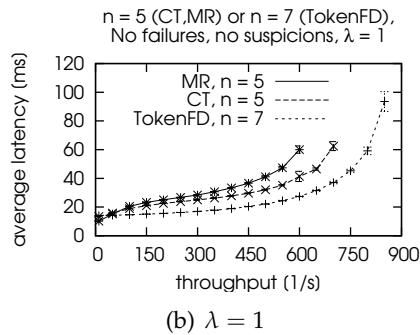
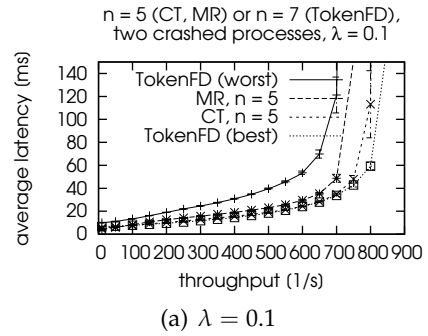
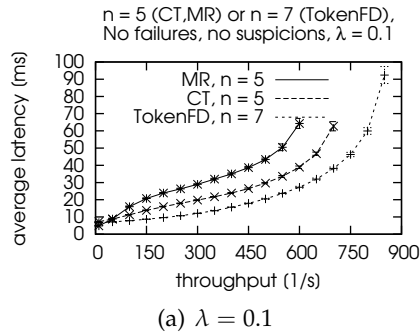


Figure 4.9: Latency vs. throughput with a *normal-steady* faultload, $n = 5$ (CT, MR) and $n = 7$ (TokenFD) correct processes

Figure 4.10: Latency vs. throughput with a *crash-steady* faultload, two crashed processes (in a group of $n = 5$ (CT, MR) and $n = 7$ (TokenFD) processes)

4.1. Token and failure detector based atomic broadcast

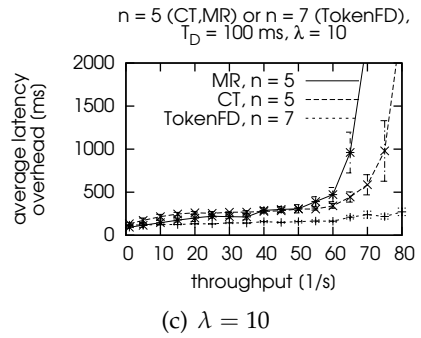
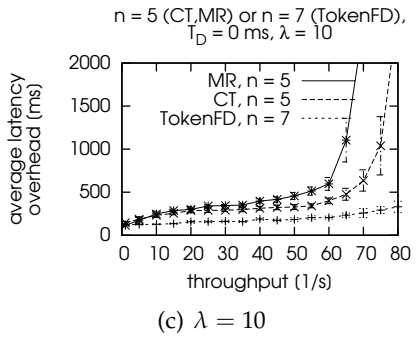
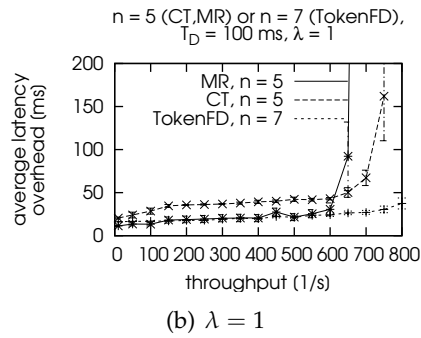
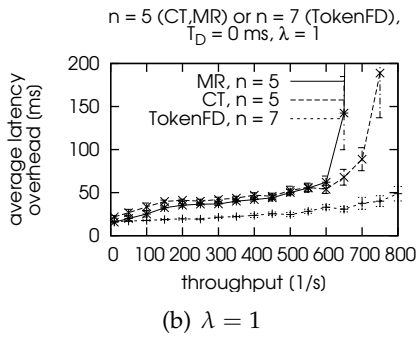
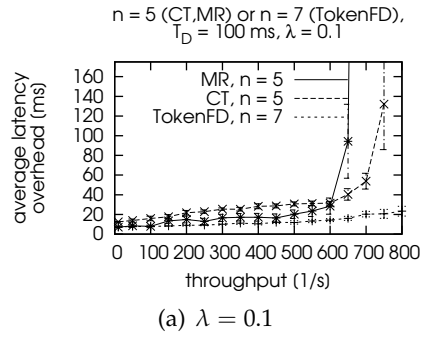
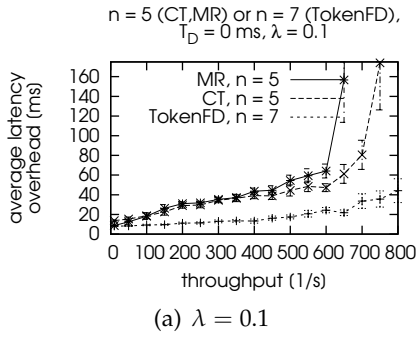


Figure 4.11: Latency overhead vs. throughput with a *crash-transient* faultload, two crashes (in a group of $n = 5$ (CT, MR) and $n = 7$ (TokenFD) processes), detection time $T_D = 0$ ms

Figure 4.12: Latency overhead vs. throughput with a *crash-transient* faultload, two crashes (in a group of $n = 5$ (CT, MR) and $n = 7$ (Token) processes), detection time $T_D = 100$ ms

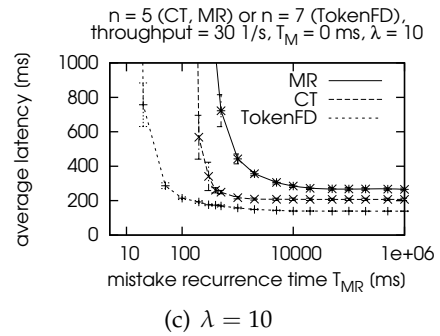
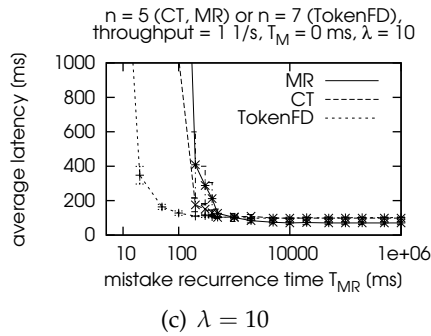
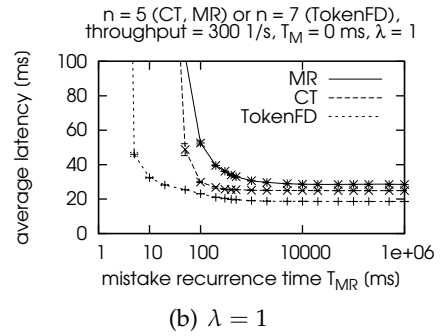
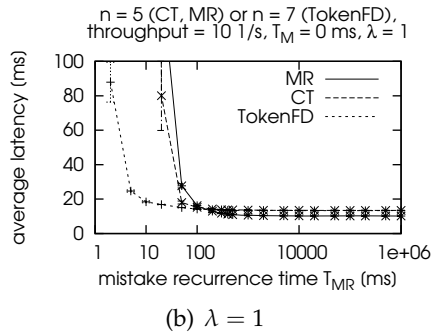
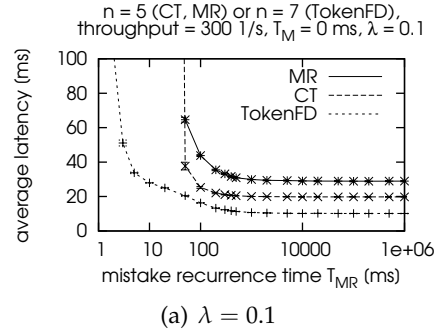
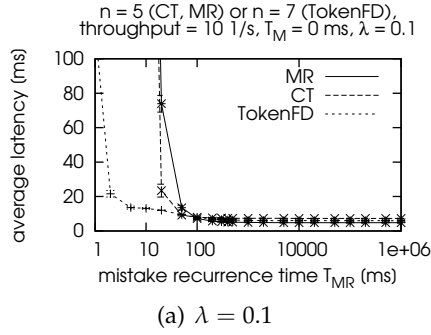


Figure 4.13: Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload, $n = 5$ (CT, MR) and $n = 7$ (TokenFD) processes, mistake duration $T_M = 0ms$, throughput of 10 ($\lambda = 0.1, 1$) or 1 ($\lambda = 10$) *abroadcasts* per second

Figure 4.14: Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload, $n = 5$ (CT, MR) and $n = 7$ (TokenFD) processes, mistake duration $T_M = 0ms$, throughput of 300 ($\lambda = 0.1, 1$) or 30 ($\lambda = 10$) *abroadcasts* per second

of number of messages, but slightly worse (compared to *MR*) in terms of communication steps. Figure 4.9 shows that in the *normal-steady* faultload, the *TokenFD* algorithm performs better than *CT* and *MR* for all values of λ , except in the case of very low loads. Again, the token-passing mechanism (and overall low message complexity) of the *TokenFD* algorithm leads to low contention on the network and the processor, which in turn leads to good simulated performance figures.

In the case of two failures, however, the results are slightly different as illustrated by Figure 4.10. In the case of the *CT* and *MR* algorithms, the (non-coordinator) process crashes are symmetrical and lead to reduced contention, thus improving the performance of the algorithms. In the case of the *TokenFD* algorithm, the position of the two crashed processes in the token ring determines the performance hit on the algorithm. Indeed, if the two crashed processes are side-by-side (best case scenario, noted "*TokenFD* (best)" in Figure 4.10), the performance hit is low: the votes in the token are reset a single time during the token circulation. Instead of deciding up to $n = 7$ times per revolution, the *TokenFD* algorithm only decides up to 3 times. If the two crashed processes have f correct processes between them (worst case scenario, noted "*TokenFD* (worst)" in Figure 4.10), the votes in the token are reset twice, precisely when f votes have already been accumulated and thus, a single decision can be taken during each token revolution. Figure 4.10 illustrates this: in the worst case scenario, the *TokenFD* performance is worse than *CT* and *MR*, whereas in the best case scenario, *TokenFD* achieves slightly lower latencies than *CT* and *MR*.

The performance graphs of the runs with a *crash-transient* faultload (with a detection time $T_D = 0$ in Figure 4.11 and $T_D = 100ms$ in Figure 4.12) show characteristics that are similar to the runs in a failure free system. With this faultload, the *TokenFD* algorithm achieves lower latencies than the other algorithms. Notice that in the crash-transient faultload, we only considered the impact on performance after a single crash.

Finally, in runs with a *suspicion-steady* faultload (wrong suspicions), illustrated in Figures 4.13 and 4.14, the *TokenFD* algorithm performs better than *CT* and *MR* as the mistake recurrence time T_{MR} decreases (more frequent wrong suspicions).

To wrap up, the simulation results show that the *TokenFD* algorithm is a better alternative to other failure detector based algorithms in various system settings, especially in the case $f = 1$ (and except at the lowest loads). In such a case, according to the simulation results, the *TokenFD* algorithm achieves lower latencies than both other algorithms, while reaching higher throughput levels. This is also the case when up to $f = 2$ failures are supported, but no crashes (nor wrong suspicions) occur. In the case of crashes in the case $f = 2$, the location of the crashed processes in the token ring largely determines the impact of the crashes on the performance of the algorithm.

4.1.7 Related work

As was mentioned in Section 4.1.1, previous atomic broadcast protocols based on tokens need group membership or an equivalent mechanism. In Chang and Maxemchuk's Reliable Broadcast Protocol [CM84], and its newer variant [MS01], an ad-hoc reformation mechanism is called whenever a host fails. Group membership is used explicitly in other atomic broadcast protocols such as Totem [AMMS⁺95], the Reliable Multicast Protocol by Whetten et al. [WMK94] (derived from [CM84]), and in [CMA97].

These atomic broadcast protocols also have different approaches with respect to message broadcasting and delivery. In [CM84, WMK94], the *moving sequencer* approach is used: any process can broadcast a message at any time. The token holder then orders the messages that have been broadcast. Other protocols, such as Totem [AMMS⁺95] or On-Demand [CMA97] on the other hand use the *privilege based* approach, enabling only the tokenholder to broadcast (and simultaneously order) messages.

Finally, the different token based atomic broadcast protocols deliver messages in different ways. In [CMA97], the token holder issues an "update dissemination message" which effectively contains messages and their global order. A host can deliver a message as soon as it knows that previously ordered messages have been delivered. "Agreed delivery" in the Totem protocol (which corresponds to *adeliver* in the protocol presented in this chapter) is also done in a similar way. On the other hand, in the Chang-Maxemchuk atomic broadcast protocol [CM84], a message is only delivered once $f + 1$ sites have received the message. Finally, the Train protocol presented in [Cri91] transports the ordered messages in a token that is passed among all processes (and is in this respect related to the token based protocols presented in this chapter).

Larrea *et al.* [LAF99] also consider a logical ring of processes, however with a different goal. They use a ring for an efficient implementation of the failure detectors $\diamond\mathcal{W}$, $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ in a partially synchronous system.

4.1.8 Discussion

According to various authors, token based atomic broadcast algorithms are more efficient in terms of throughput than other atomic broadcast algorithms. The reason is that the token can be used to reduce network contention. However, all published token based algorithms rely on a group membership service, *i.e.*, none of them use unreliable failure detectors directly. The first part of this chapter presented the first token based atomic broadcast algorithms that solely relies on a failure detector, namely the new failure detector called \mathcal{R} . Such an algorithm has the advantage of tolerating failures *directly* (*i.e.*, it also tolerates wrong failure suspicions). Algorithms that do not tolerate failures directly, need to rely on a membership service

to exclude crashed processes. As a side-effect, these algorithms also exclude correct processes that have been incorrectly suspected. Thus, failure detector based algorithms have advantages over group membership based algorithms, in case of wrong failure suspicions, and possibly also in the case of real crashes.

Finally, although token based atomic broadcast algorithms are usually considered to be efficient only in terms of throughput, our performance evaluation has shown that for small values of n , our algorithm compares favorably with the Chandra-Toueg atomic broadcast algorithm (using the Chandra-Toueg or Mostéfaoui-Raynal consensus algorithm) in terms of latency as well, at all but the lowest loads.

Finally, the performance comparison in this chapter was based on simulated results. The second part of this thesis focuses on the experimental performance evaluation of atomic broadcast algorithms. As a consequence, the experimental evaluation of our new algorithm and the comparison with the Chandra-Toueg algorithm and a token based algorithm using group membership is presented in Chapter 7. Chapters 8 and 9 also present experimental performance results of our new token and failure detector based algorithm.

4.2 Variants and optimizations of the token based algorithm

In the previous section, we presented a novel token-passing atomic broadcast algorithm that uses a failure detector. The *Uniform agreement* property (which corresponds to termination) of atomic broadcast in the algorithm presented in Section 4.1.5.B is ensured by passing the *entire* set of consensus decisions in the token: whenever a process receives the token, it can catch up on all decisions taken up to that point. Although the algorithm is correct, this is of course extremely inefficient in practice. Realistically, the token based algorithm needs to be implemented so that data structures that are sent on the network do not grow indefinitely (or at least, that grow very slowly, *i.e.*, logarithmically).

With this constraint in mind, we now modify the token based atomic broadcast algorithm in order to bound the size of the set of decisions transported in the token, without affecting the *Uniform agreement* of atomic broadcast: *If a process p delivers m , then all correct processes eventually deliver m .*

Specifically, we focus on the case where a correct process *delivers* m and show how to guarantee that all correct processes eventually *deliver* m (from the algorithm and the properties of the failure detector \mathcal{R} , it is easy to show that if a faulty process *delivers* m , then some correct process eventually *delivers* m). Ensuring that all correct process *deliver* m once

one of them has *delivered* m is easy, but allows several alternatives. These alternatives are discussed in the following section.

Let us assume that a correct process *delivers* m as part of decision number d_m . We show how to ensure that all correct processes eventually receive decision number d_m and discuss the effects on the amount of memory needed, the number of backup tokens that are sent and the diffusion of new decisions. We start by examining the case where the process p_i can locally store an unbounded set $deliv_i$ (Section 4.2.1) and then show how to achieve similar properties while limiting the size of the $deliv_i$ sequence (Section 4.2.2). These results are summarized in Table 4.1 (page 64).

4.2.1 Unbounded $deliv_i$ set of ordered messages

Each process p_i has an $deliv_i$ set of consensus decisions containing messages that have been delivered. If we assume that this sequence can grow indefinitely, then the *Agreement* property of atomic broadcast is ensured as follows. All processes keep a copy of each consensus decision that is taken. If a process detects that a decision is missing (upon receiving a token containing a decision with a number that is higher than expected for example), it sends a decision request to all processes. A process that receives such a request and has a subset of the requested decisions replies to the sender and includes the asked-for decisions.

This approach is one of those currently implemented and is close to Algorithms 4.5 and 4.6. It needs an ad-hoc retransmission mechanism and an unbounded $deliv_i$ set. The backup tokens (*i.e.* those sent by p_i to $p_{i+2}, \dots, p_{i+f+1}$) are not necessary as long as there aren't any suspicions, but are sent if a request is received from a process p_k suspecting its predecessor p_{k-1} . As in Algorithm 4.6, decisions do not need to be broadcast (using reliable broadcast or send-to-all) but are instead diffused within the token.

Proof. Let a correct process p reach decision number d_m . From the token circulation and the properties of the failure detector \mathcal{R} , it is easy to show that all correct processes eventually receive a token with a decision number $d_t \geq d_m$. Let q be a correct process that has not decided in consensus d_m . Upon receiving a token with $d_t \geq d_m$ either (i) $d_t = d_m$ and thus q reaches a decision in consensus d_m or (ii) $d_t > d_m$ and q sends a request message to all for all decisions whose number is smaller than d_t and that q has not received. By the properties of quasi-reliable channels, q eventually receives the decision taken in instance d_m from p . \square

4.2.2 Bounded-size $adeliv_i$ set of ordered messages

In the Chandra-Toueg atomic broadcast algorithm, a process p_i can garbage collect all decisions whose messages have already been *adelivered*. This can be done because each decision is reliably broadcast to all processes (thus, if a correct process receives a decision, all correct processes eventually receive it). In Algorithm 4.6 and in the case of the approach presented in the previous paragraph, the consensus decisions are conveyed in the token. When backup tokens are not sent, it is impossible to determine which correct processes have received all batches of ordered messages (*i.e.*, the consensus decisions) and thus, the $adeliv_i$ set of a process p_i cannot be bounded (since any previous consensus decision can potentially be requested by a correct process).

On the other hand, if the consensus decisions of the token based atomic broadcast algorithm are reliably broadcast, the size of the $adeliv_i$ set can be bounded, as in the case of the Chandra-Toueg atomic broadcast algorithm. This result is presented in the following paragraphs. After that, a second approach that does not need decisions to be broadcast is also presented.

4.2.2.A Reliable broadcast of new decisions

The first variant of the token based atomic broadcast algorithm that allows to bound the set of consensus decisions that need to be stored inside the token and by the processes is the following. Whenever a decision is taken, the token holder reliably broadcasts the decision to all processes. Consequently, the first time a process receives a decision, it is either (1) in the token or (2) in an *rdeliver*. In case (1), whenever a process receives a decision in the token that has not yet been *rdelivered*, it *rbroadcasts* that decision to all processes: this is necessary in order to ensure that all correct processes eventually *rdeliver* all decisions. Otherwise, a faulty process could decide a value, send it in the token and crash before *rbroadcasting* it: the decision in the token would never be *rdelivered*. When a process has *rdelivered* a decision and processed it, the decision can be garbage collected.

This approach requires at least one reliable broadcast per decision that is taken. However, it removes the need for sending backup tokens in good runs and avoids having to save the entire set of consensus decisions in the token.

Proof. With the reliable broadcasts of new decisions, we ensure that if a correct process receives a decision, all correct processes receive it. Indeed, a correct process can receive a decision either (i) in an *rdeliver*, (ii) by being the first process that takes that decision or (iii) by receiving the decision in a token. In case (i), all correct processes eventually *rdeliver* the decision. In cases (ii) and (iii), the correct process *rbroadcasts* the decision and thus, all correct processes eventually *rdeliver* it. \square

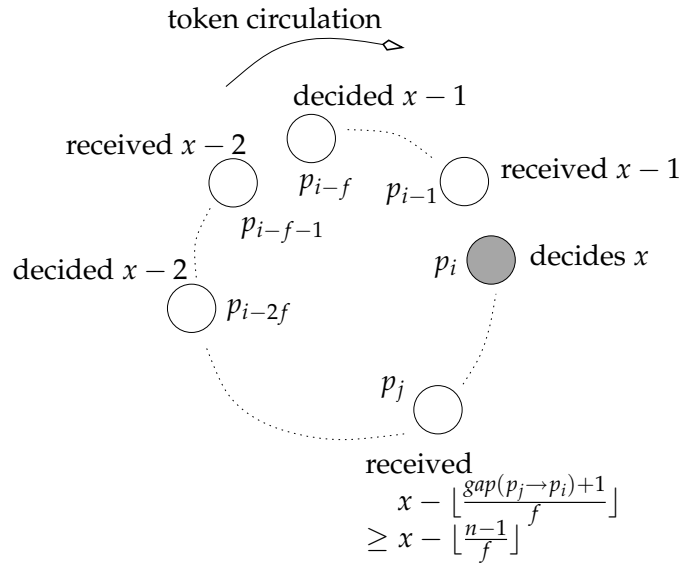


Figure 4.15: Illustration of Lemma 4.2.2: the token holder, p_i , takes decision number x . All processes have received at least decision number $x - \lfloor \frac{n-1}{f} \rfloor$ in the same round.

4.2.2.B No reliable broadcast of the decision

The second approach that allows us to bound the set of consensus decisions that need to be stored by all processes does not require decisions to be *rbroadcast* (or broadcast at all for that matter). It assumes that all correct processes receive at least one token in each round. This is achieved by forcing a token holder to always send the token to its $f + 1$ neighbors (*i.e.*, *always* send the backup tokens).

Let's analyze the situation where a process p_i takes decision number x in round r . We claim that all processes have received at least decision number $x - \lfloor \frac{n-2}{f} \rfloor - 1$, either in round r or in round $r - 1$. The result is shown in two parts:

Lemma 4.2.1. *We assume that decisions are only transported in the token. If p_i decides x in round r , then p_{i-f}, \dots, p_{i-1} eventually receive at least decision $x - 1$ in that round.*

Proof. Just before deciding in round r , either (1) p_i receives a token with decision $x - 1$, reaches the vote threshold $f + 1$ and is the first process to decide x , or (2) p_i receives a token that already contains decision x (and some other process decided x before p_i). In case (1), p_i receives a token with f votes. Processes p_{i-f} to p_{i-1} have thus voted for the future decision x and have received decision $x - 1$ in the token. In case (2), p_i receives a token

which contains decision x in round r . From the algorithm, all processes in p_{i-f}, \dots, p_{i-1} eventually receive a token with decision x or $x - 1$. \square

Lemma 4.2.2. *Let p_i be a process that has taken decision number x . All other processes then eventually receive a token with decision number $x - \lfloor \frac{n-1}{f} \rfloor$.*

Proof. Let us assume that process p_i takes decision x in round r . From Lemma 4.2.1, processes p_{i-f}, \dots, p_{i-1} (and in particular p_{i-f}) eventually receive the decision $x - 1$ in that round. By induction, the set of processes $p_{i-kf}, \dots, p_{i-(k-1)f-1}$, with $k \in [1, \lceil \frac{n}{f} \rceil]$, eventually receive decision $x - k$. In particular $p_{i-n+1} = p_{i+1}$, eventually receives at least decision $x - \lfloor \frac{n-1}{f} \rfloor$ in round r if $p_i = p_{n-1}$, round $r - 1$ otherwise. \square

Figure 4.15 illustrates Lemma 4.2.2: process p_i , the token holder, is the first process to take decision number x (in round r). Processes p_{i-f} to p_{i-1} have thus received decision $x - 1$. In general, any process, by the end of round r , has received at least decision number $x - \lfloor \frac{n-1}{f} \rfloor$.

If multiple proposals are allowed in the same token, then whenever a decision x is taken, all other processes will at least receive a token with decision $x - n + 1$.

Impacts on garbage collection After the x^{th} decision in round r and since all processes receive a token in a given round, all processes have at least decision $x - \lfloor \frac{n-1}{f} \rfloor$ (or $x - n + 1$ if multiple proposals are used) after round r . The process that takes the decision can thus garbage collect all decisions with numbers up to $x - \lfloor \frac{n-1}{f} \rfloor$. Similarly, a process that receives a token with decision x can garbage collect decisions with numbers smaller than $x - \lfloor \frac{n-1}{f} \rfloor$.

4.2.2.C Summary

Table 4.1 summarizes the three approaches that allow the token based atomic broadcast algorithm to limit the size of the tokens that are sent, while ensuring that the *Uniform agreement* property of atomic broadcast holds.

1. If the processes have an unbounded amount of memory, *Uniform agreement* is guaranteed by storing all known decisions locally and retransmitting all decisions that are requested by other processes. This approach has the following advantages: a process can skip rounds $[r + 1, r' - 1]$ if it receives a token from round $r' > r + 1$ where r is the round of the previously held token. Furthermore, backup tokens are not needed in runs without failures or suspicions, which is one of the optimizations described in Section 4.1.5.C. Finally, decisions do not need to be broadcast as soon as they are taken. This solution is the one that requires the least number of messages.

Table 4.1: Summary of the different approaches that ensure the *Uniform agreement* property of atomic broadcast

	Unbounded memory	Bounded memory
Backup tokens always sent	—	3. Rounds cannot be skipped, new decisions don't need to be broadcast
No backup tokens in good runs	1. Retransmission mechanism for lost decisions, new decisions don't need to be broadcast	2. New decisions must be reliably broadcast

2. If the processes have a bounded memory, then decisions can be reliably broadcast instead of including all of them in the token. This approach also avoids sending backup tokens, but requires (costly) reliable broadcasts of all decisions. This approach is the closest in terms of message complexity to the Chandra-Toueg atomic broadcast and consensus algorithms, where consensus decisions are also reliably broadcast.
3. Finally, if processes have a bounded memory and reliable broadcast cannot be used, the third approach is to *always* send the f backup tokens. With this approach, the decisions no longer need to be broadcast and the token only needs to transport the n last decisions that have been taken. The drawback of this approach is that processes need to receive a token in each round (by always sending the f backup tokens, we ensure that a correct process receives at least one token per round).

4.3 Adapting the algorithm to the Heard-Of model

The token based consensus (and atomic broadcast) algorithm presented in Section 4.1 uses an unreliable failure detector to ensure that a decision is eventually taken (or that a broadcast message is eventually delivered). The failure detector model approach to fault tolerance is characterized by a strong emphasis on process failures, and in particular on the fail-stop class of failures. In the case of fail-stop, a process simply stops executing processing steps and remains crashed forever. Other types of (benign) crashes such as send and receive omission failures, where a faulty process fails to send or receive some messages, cannot be handled by the failure detector model.

Furthermore, failure detector based consensus algorithms (e.g. [CT96] and [MR99]) often assume reliable (or quasi-reliable) channels and thus do not support the loss of messages on the network (link failures).

The Heard-Of model (noted HO model hereafter and presented in Section 3.1.3) addresses these problems: processes communicate in rounds (each composed of send and state transition stages). Instead of being informed on (potentially) faulty processes, a process receives a set of messages (the *heard-of* set) at the beginning of the state transition stage of each round. If a process p_i expects a message from process p_j in round r and that the message is not in the received set of round r , this loss can be explained by several reasons: p_j might have crashed or might be subject to send-omission, the message might have been lost by the network or p_i might be subject to receive-omission. In any case, the HO model does not try to interpret the loss of a message: it only provides the set of received messages, which are then used to solve consensus. The sets of received messages can of course not be arbitrary (for instance, how can one solve consensus if the received sets are empty for all rounds and all processes?) and thus, each algorithm has an associated *predicate* that imposes conditions on the heard-of sets.

In this section, our goal is to evaluate the HO model in the context of the token algorithm. We first present the token based consensus algorithm in the context of the HO model and then discuss what predicate is needed by the algorithm to ensure the *Termination* of consensus. We consider a group $\Pi = \{p_0, \dots, p_{n-1}\}$ of n processes. Each phase ϕ of the algorithm has n rounds $r = \{n\phi, \dots, n(\phi + 1) - 1\}$ (i.e. $r - n\phi = r \bmod n \in \{0, \dots, n - 1\}$). Rounds start at $r = 1$.

4.3.1 Description of the algorithm

In the HO model, algorithms proceed in rounds consisting of sending and state transition stages. For the sake of clarity, rounds can then be grouped into *phases*. In the token based consensus algorithm, processes pass a token along a logical ring and we associate a phase to a complete revolution of the token. Each phase thus naturally corresponds to n consecutive rounds.¹⁰

Furthermore, in each phase, a process only receives tokens in one round (in phase k , the reception round of p_i is $k \cdot n + i$) but sends tokens in $f + 1$ rounds (i.e. one round for sending the logical token and f rounds for sending the backup tokens). The dual approach (sending in a single round and receiving in $f + 1$ rounds) is also possible, but yields an algorithm that is slightly longer.

Algorithm 4.7 presents the token accumulation consensus algorithm in

¹⁰In the token based consensus algorithm, a round in the failure detector model (Section 4.1.4.C) corresponds to a phase in the HO model...

Algorithm 4.7: Token accumulation consensus in the *Heard-Of* model
(code of process p_i)

```

1: upon  $propose(v_i)$  do
2:    $estimate_i \leftarrow v_i$ 
3:    $decided_i \leftarrow \text{false}$ 
4:    $votes_i \leftarrow 1$ 
5:    $tokenToSend_i \leftarrow \langle 1, estimate_i, \text{false} \rangle$ 

6: Round  $r$  (phase  $\phi = \lfloor \frac{r}{n} \rfloor$ ):
7:    $S^r$  :
8:     if  $r - n\phi \in \{(i + 1) \bmod n, \dots, (i + f + 1) \bmod n\}$  then
9:       send  $tokenToSend_i$  to  $p_{r-n\phi}$            {Send token to the receiver  $p_{r \bmod n}$ }

10:   $T^r$  :
11:    if  $i \neq r - n\phi$  or  $decided_i$  then           {Not a reception round or already decided.}
12:      skip  $T^r$ 
13:    if received  $\langle -, estimate, \text{true} \rangle$  then           {if one of the tokens has a decision}
14:       $decided_i \leftarrow \text{true}$ 
15:       $estimate_i \leftarrow estimate$ 
16:       $decide(estimate_i)$ 
17:    else
18:       $p_k \leftarrow$  process  $p_j$  with the smallest  $gap(p_j \rightarrow p_i)$  s.t. received
19:         $\langle votes, estimate, \text{false} \rangle$  from  $p_j$            {adopt the estimate in the token}
20:       $estimate_i \leftarrow estimate$ 
21:      if  $p_k = p_{(i-1) \bmod n}$  then
22:         $votes_i \leftarrow votes + 1$            {token received from predecessor: increment votes}
23:      else
24:         $votes_i \leftarrow 1$            {gap in token circulation: reset votes}
25:      if  $votes_i \geq f + 1$  then           {if enough votes have been gathered: decide}
26:         $decided_i \leftarrow \text{true}$ 
27:         $decide(estimate_i)$ 
28:       $tokenToSend_i \leftarrow \langle votes_i, estimate_i, decided_i \rangle$            {Save the token to send}

```

4.3. Adapting the algorithm to the Heard-Of model

Table 4.2: Illustration of which processes send and receive messages in a given round of the token based consensus algorithm in the Heard-Of model

<i>phase</i>	<i>round</i>	S^r	T^r
$\phi = 0$	$r = 0$ <i>skipped</i>	-	-
	$r = 1$	p_2, p_0	p_1
	$r = 2$	p_0, p_1	p_2
$\phi = 1$	$r = 3$	p_1, p_2	p_0
	$r = 4$	p_2, p_0	p_1
	$r = 5$	p_0, p_1	p_2
\vdots	\vdots	\vdots	\vdots
$\phi = k$	$r = n \cdot k$	p_1, p_2	p_0
	$r = n \cdot k + 1$	p_2, p_0	p_1
	$r = n \cdot k + 2$	p_0, p_1	p_2

the HO model. During the initialization (upon proposing a value) process p_i sets its estimate to its own proposal (line 2) and the $decided_i$ flag to false (line 3).

For each round r , process p_i then first executes the send stage S^r (lines 7 to 9). In this stage, p_i checks if r is the reception round of one of its $f + 1$ successors (line 8) and if so sends the token that was previously saved (line 9).

In the state transition stage T^r of round r (lines 10 to 27), p_i starts by verifying if r is its own reception round ($r = n\phi + i$) on line 11. If this is not the case T^r is skipped. T^r is also skipped if p_i has decided: in this case, the variable $tokenToSend_i$ contains the decided value and should not be modified.

If p_i has not decided and r is its reception round, p_i examines its received set (line 13): either (1) a token with a decision has been received (lines 14 to 16) or (2) only regular tokens were received (lines 18 to 26). In the first case (1), p_i sets its $decided_i$ flag, adopts the estimate in the token and decides (line 16). In the second case (2), p_i selects the token that was received from the process with the smallest gap on line 18 (and adopts the estimate contained in the token on line 19). Process p_i then increments the votes if the token was received from the immediate predecessor (line 21) and resets the votes otherwise (line 23). If enough votes have been collected, p_i decides (lines 24 to 26). Finally, if a token was received, p_i sets $tokenToSend_i$ with the new values of $votes_i$, $estimate_i$ and the decision flag $decided_i$ (line 27). This token is then sent in the sending stage of the next $f + 1$ rounds.

Table 4.2 shows which processes send (S^r column) and receive (T^r column) tokens in a given round in a system with $n = 3$ processes. Notice that, as explained above, there is a single receiving process per round (but $f + 1 = 2$ senders) and that each phase $\phi > 0$ corresponds to $n = 3$ rounds: a complete token revolution.

4.3.2 Heard-Of predicates

A number of conditions, defined by predicates on the sets of messages received in each round, must be met for Algorithm 4.7 to solve consensus in the HO model. These conditions form the *predicate*. We now present the predicates which ensure that the algorithm above satisfies the *Uniform agreement* and *Termination* properties of consensus. Algorithm 4.7 also satisfies the *Uniform validity* and *Uniform integrity* properties of consensus, which is easy to show and omitted here.

4.3.2.A Uniform agreement

Algorithm 4.7 above ensures *Uniform agreement* of consensus (no two processes decide a different value) without the need for any predicate.

Proof. The proof of *Uniform agreement* is identical to the corresponding proof of the failure detector based version of the token algorithm (presented in Section 4.1.4.C on page 38). \square

4.3.2.B Termination

As mentioned earlier, Algorithm 4.7 needs a predicate to ensure that consensus eventually terminates (otherwise, the FLP impossibility result would be violated). The following section discusses how the predicate for *Termination* is derived.

Note on token circulation We start by presenting a predicate $\mathcal{P}_{tokensForever}$ on the token circulation. The predicate states that each process must receive at least one message (a token) in its reception round (which, for a process p_i is all rounds such that $r \bmod n = i$). This predicate is sufficient, but not necessary, to ensure that there is always at least one token in circulation.

$$\mathcal{P}_{tokensForever} \equiv \forall r : HO(p_{r \bmod n}, r) \neq \emptyset$$

\mathcal{P}_{token} predicate The $\mathcal{P}_{tokensForever}$ predicate only ensures that tokens circulate forever, which is not sufficient for eventually reaching a decision. The following predicate, \mathcal{P}_{token} (which is stronger than $\mathcal{P}_{tokensForever}$) ensures *Termination* of Algorithm 4.7.

$$\begin{aligned} \mathcal{P}_{token} \equiv & \forall r : HO(p_{r \bmod n}, r) \neq \emptyset \text{ and} \\ & \exists \Pi_f \subseteq \Pi : |\Pi_f| \geq n - f \text{ and } \exists \phi : \forall \phi' \in [\phi, \phi + 2] : \forall p_i \in \Pi : \\ & HO(p_i, i + n\phi') = \Pi_f \cap \{p_{(i-f-1) \bmod n}, \dots, p_{(i-1) \bmod n}\} \end{aligned}$$

The \mathcal{P}_{token} predicate ensures (1) that $\mathcal{P}_{tokensForever}$ holds and (2) that there is a set of processes Π_f of cardinality $n - f$ or greater and a phase ϕ after which the system ensures the following: for all phases $\phi' \in [\phi, \phi + 2]$, each process p_i receives a message during its reception round $(i + n\phi')$ from all of its predecessors that are in Π_f . This set Π_f remains unchanged for the three phases $[\phi, \phi + 2]$.

For example, consider a system with $n = 7$ processes $\{p_0, \dots, p_6\}$ and $f = 2$. If $\Pi_f = \{p_0, p_1, p_3, p_4, p_6\}$ and $\phi = 6$, then the predicate ensures that for all values of ϕ' between 6 and 8, we have, for p_0 and p_2 , $HO(p_0, 7\phi') = \{p_4, p_6\}$ and $HO(p_2, 2 + 7\phi') = \{p_6, p_0, p_1\}$.

The \mathcal{P}_{token} predicates, with $n \geq f(f + 1) + 1$, allows Algorithm 4.7 to ensure the *Termination* property of consensus.

Proof. We assume that $n \geq f(f + 1) + 1$. Consequently, $|\Pi_f| \geq n - f \geq f^2 + 1$. Π_f is formed of at most f sets of consecutive processes. One of these sets thus contains $f + 1$ processes:

$$\exists p_i \text{ s.t. } \{p_i, \dots, p_{(i+f) \bmod n}\} \subseteq \Pi_f \quad (4.1)$$

We now show that a decision is taken at the latest in phase $\phi + 1$ by process $p_{(i+f) \bmod n}$. First of all, the $\mathcal{P}_{tokensForever}$ predicate ensures that a least one token circulates until phase ϕ . Then, from Equation 4.1 and \mathcal{P}_{token} , we have

$$\begin{aligned} & \forall k \in 0, \dots, f - 1 : p_{(i+k) \bmod n} \in \Pi_f \cap \{p_i, \dots, p_{(i+f) \bmod n}\} \\ \Rightarrow & \forall k \in 0, \dots, f - 1 : p_{(i+k) \bmod n} \in HO(p_{(i+k+1) \bmod n}, i + k + 1 + \phi \cdot n) \end{aligned}$$

Let us assume that p_i sets $tokenToSend_i$ to $\langle 1, -, - \rangle$ in round $i + \phi \cdot n$. Processes $p_{(i+1) \bmod n}$ to $p_{(i+f) \bmod n}$ receive a token from from their immediate predecessors in rounds $i + 1 + \phi \cdot n$ to $i + f + \phi \cdot n$ and all execute line 21. Consequently, $p_{(i+f) \bmod n}$ sets $votes_i$ to $f + 1$ and decides (lines 25 and 26) in round $i + f + \phi \cdot n$ (which is in phase ϕ if $i + f < n$, $\phi + 1$ otherwise).

From the \mathcal{P}_{token} predicate and since $p_{i+f} \in \Pi_f$, it is easy to show that all other processes receive a token with a decision in one of the phases ϕ to $\phi + 2$. \square

4.3.3 Discussion

The predicate $\mathcal{P}_{tokensForever}$ is weaker than the requirement of quasi reliable channels of the failure detector solution. However, $\mathcal{P}_{tokensForever}$ requires

a stronger assumption than fair-lossy channels. A detailed quantitative analysis is needed to determine whether the solution expressed in the HO model presents, for the token algorithm, a real advantage over the solution based on failure detectors.

Solving atomic broadcast with indirect consensus

As seen in the previous chapters, atomic broadcast and consensus are important abstractions in fault tolerant distributed computing. In [CT96], the authors present a reduction of atomic broadcast to consensus. In this reduction, the atomic broadcast algorithm performs consensus runs on sets of messages in order to determine the delivery order of those messages.

While this is correct from a theoretical point of view, it is inefficient in practice. Indeed, executing consensus on messages can lead to heavy network usage if the messages are large. Instead, if consensus is executed on message identifiers (*indirect* consensus), the messages themselves only need to be diffused once and the ordering process is then done on light-weight message identifiers.

Executing consensus on message identifiers has already been done in previous group communication implementations [Fel98, UDS02] and has always been seen as being easy, given a consensus algorithm on messages. However, in these group communication implementations, the consensus algorithms were not adapted to handle message identifiers instead of messages. As a consequence, if at least one process can crash, it can lead to a faulty execution, as we show in this chapter.

To correctly implement a group communication stack, the consensus and atomic broadcast algorithms need to be adapted to the case where the decision is taken on identifiers instead of messages. We show that these modifications are not trivial for all consensus algorithms and can affect their resilience.

Contributions We start by discussing and illustrating the advantages of executing consensus *indirectly* on message identifiers rather than on messages. Two contributions are then presented in this chapter: we (1) start by presenting *indirect* consensus, and show what guarantees it must pro-

vide to ensure the correctness of the atomic broadcast algorithm. We then (2) show that the transformation of failure-detector based consensus algorithms on messages into *indirect* consensus algorithms on message identifiers is far from trivial: the resilience (*i.e.* the number of supported failures) of some consensus algorithms can be affected by the modifications. To illustrate this, two $\diamond\mathcal{S}$ -based consensus algorithms (proposed by Chandra and Toueg, and Mostéfaoui and Raynal respectively) are adapted into indirect consensus algorithms that work on message identifiers. The resilience of one of the algorithms is affected by the modifications whereas the other one isn't.

The chapter is structured as follows. In Section 5.1, we motivate the use of consensus on message identifiers rather than on messages and present the formal specification of *indirect* consensus. Section 5.2 illustrates the modifications that are needed to transform two consensus algorithms with the failure detector $\diamond\mathcal{S}$ into indirect consensus algorithms. The section emphasizes the fact that not all consensus algorithms on messages can be trivially modified into indirect consensus algorithms on message identifiers. Section 5.3 compares the performance of the consensus and indirect consensus algorithms presented in Section 5.2. Finally, Section 5.4 concludes this chapter.

5.1 Motivation and indirect consensus

5.1.1 Atomic broadcast on message identifiers

In the following paragraphs, we discuss the use of message identifiers in atomic broadcast algorithms. We start by giving a short reminder of the specifications of reliable and atomic broadcast. We then recall the reduction of atomic broadcast to consensus and show performance comparisons between executions of atomic broadcast using messages and executions using message identifiers. The specifications and the short reminder on the reduction of atomic broadcast to consensus help in understanding the problems involved when executing consensus on message identifiers.

We consider an asynchronous system composed of n processes taken from a set $\Pi = \{p_0, \dots, p_{n-1}\}$. The processes communicate by passing messages over reliable channels and can only fail by crashing (no Byzantine failures). A process that never crashes is said to be *correct*, otherwise it is *faulty*.

We now give an informal reminder of the agreement problems that are considered in this chapter. The formal definitions of the problems are presented in Section 3.2.

Informally, reliable broadcast guarantees that all correct processes deliver the same set of messages. The (uniform) atomic broadcast problem

5.1. Motivation and indirect consensus

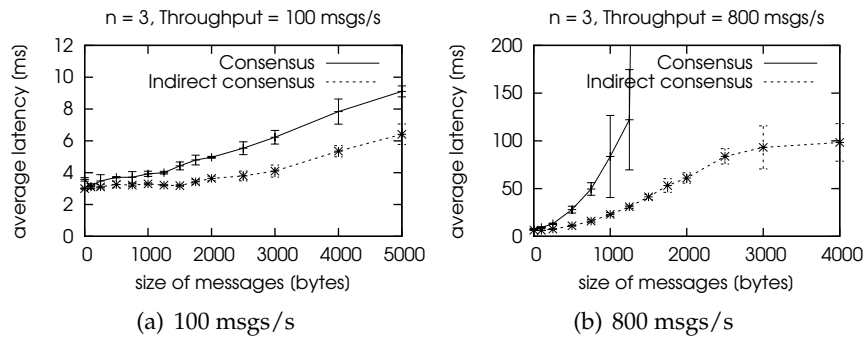


Figure 5.1: Latency vs. message size in a system with 3 processes

is reliable broadcast augmented with an additional total order property on the delivered messages. Finally, consensus allows a group of processes to reach a common decision.

In [CT96], the authors present a reduction of atomic broadcast to consensus. In this reduction, whenever a message m is *abroadcast*, m is reliably broadcast to all processes. Following this, whenever a process receives a message that it hasn't already *adelivered*, it executes consensus in order to reach a decision with the other processes on the next message to *adeliver*. In the reduction of atomic broadcast to consensus in [CT96], the processes thus execute a sequence of consensus runs on sets of messages as long as new messages are *abroadcast*.

This reduction is correct. However, in a system where the messages are *large*, the consensus runs are executed on sets of *large* messages. Thus, the size of the data exchanged by the consensus algorithm is large and can

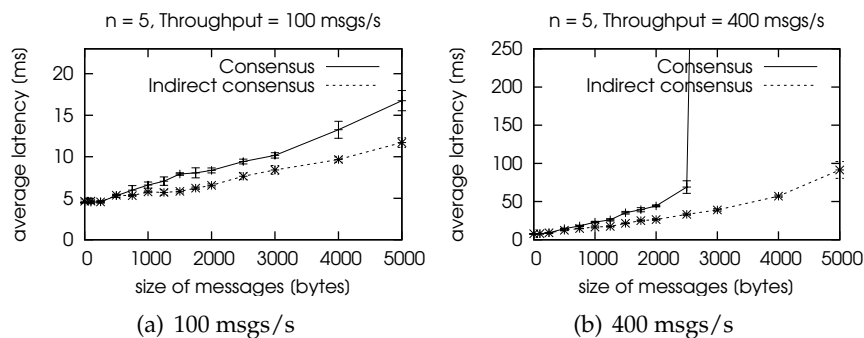


Figure 5.2: Latency vs. message size in a system with 5 processes

potentially saturate the system. In order to avoid this, consensus can be executed on message identifiers instead of the messages themselves. This decouples the size of the messages from the size of the data exchanged by consensus. Since the relationship between the messages and their identifiers is bijective, the delivery order of the messages can easily be inferred from the ordered sequence of message identifiers (which ensures the *Uniform total order* property of atomic broadcast).

The performance gain when using message identifiers instead of messages in consensus is not negligible. Indeed, the size of a message identifier is independent of the size of the message itself. Thus, the size of the data exchanged by consensus remains constant as the size of the messages increases. Figures 5.1 and 5.2 illustrates the performance difference between executing consensus on messages or message identifiers in the context of atomic broadcast. The performance metric for atomic broadcast is the *average latency*, defined as the average (over all processes) of the elapsed time between *abroadcasting* a message m and *adelivering* m . The figure shows the latency of atomic broadcast as a function of the size of the messages. The results are shown for two *throughputs* (the overall rate of atomic broadcasts in the system: 100 or 800 *msgs/s* for $n = 3$, 100 or 400 *msgs/s* for $n = 5$). The tests were done using the Neko framework on a local area network with Pentium III machines. More details on the framework and the system setup can be found in Section 5.3.

One can clearly see that as the size of the messages increases, the latency of consensus on message identifiers (noted *Indirect consensus* in the figure) is lower than the latency when using entire messages (noted *Consensus*). This result becomes clearer as the throughput of atomic broadcasts increases and as the size of the system increases. As a consequence, except for trivial conditions (low throughputs and small systems), executing consensus on message identifiers rather than on entire messages is clearly justified.

5.1.2 Violating the *Validity* of atomic broadcast

Executing consensus on message identifiers implies that the consensus and atomic broadcast algorithms must be adapted to explicitly handle message identifiers instead of messages, as we now show. More specifically, we show that if the atomic broadcast algorithm directly executes the original consensus algorithm in [CT96] on message identifiers, the *Validity* property of atomic broadcast could be violated. The scenario is illustrated in Figure 5.3 and is described below. Imagine that a faulty process p_0 *abroadcasts* a new message m : p_0 reliably broadcasts m and starts executing consensus on its message identifier $id(m)$. Let us assume that the consensus algorithm decides on $id(m)$, p_0 crashes and no other process receives a copy of

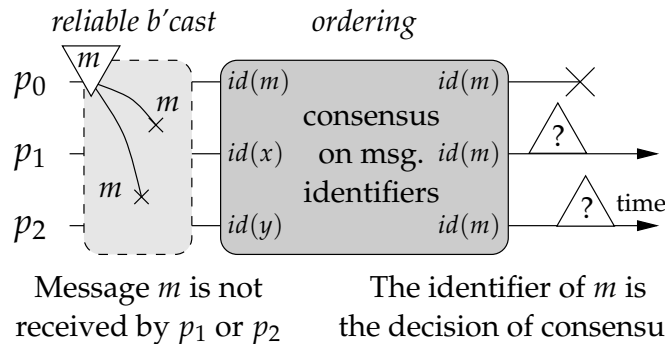


Figure 5.3: Illustration of the violation of the *Validity* of atomic broadcast if consensus is executed directly on message identifiers

m . No other process than p_0 is able to deliver m (and thus any message ordered after m). Furthermore, in order to guarantee the *Uniform total order* property of atomic broadcast, $id(m)$ cannot be removed from the sequence of ordered message identifiers. As a consequence, the *Validity* property of atomic broadcast is violated in such an execution, since m and any following message in the ordered sequence cannot be delivered.

The problem described above could be avoided by using uniform reliable broadcast instead of reliable broadcast in the atomic broadcast algorithm. Uniform reliable broadcast guarantees that if at least one process (correct or not) delivers a message, then all correct processes eventually deliver that message [HT94]. Since the atomic broadcast algorithm in [CT96] only executes consensus on messages that have been uniformly reliably delivered, this solution guarantees that all correct processes eventually receive a copy of any message ordered by consensus. However, the cost of using uniform reliable broadcast is higher than that of reliable broadcast (this is later illustrated in Section 5.3.4).

Instead of using uniform reliable broadcast and incurring its cost, we suggest to adapt the consensus algorithm to handle message identifiers and provide additional properties that ensure the correctness of atomic broadcast.

5.1.3 Indirect consensus

The motivation of introducing *indirect* consensus is to capture the differences between executing consensus on messages and on message identifiers. Instead of executing consensus directly on messages, we want to *indirectly* execute consensus on message identifiers. Simultaneously, indirect consensus has to offer guarantees to atomic broadcast so that all the messages whose identifiers have been ordered can be delivered by atomic

broadcast.

In *indirect* consensus, each proposal is a pair (v, rcv) , where v is a set of message identifiers (and $msgs(v)$ are the messages whose identifiers are in v). rcv is a function such that $rcv(v)$ returns true only if the process has received $msgs(v)$. Whenever a decision is taken on v , indirect consensus must ensure that all correct processes eventually receive $msgs(v)$. In the context of indirect consensus, we introduce the following hypothesis on the rcv function:

Hypothesis A: If $rcv(v)$ is true for a correct process, then $rcv(v)$ is eventually true for all correct processes.

Formally, we specify *indirect* consensus similarly to consensus, in terms of two primitives: $propose(v, rcv)$ and $decide(v)$. The (uniform) *indirect* consensus problem is then specified by five properties. The four first properties are (almost) identical to (uniform) consensus:

Termination If the *Hypothesis A* holds, then every correct process eventually decides some value.

Uniform integrity Every process decides at most once.

Uniform agreement No two processes (correct or not) decide a different value.

Uniform validity If a process decides v , then (v, rcv) was proposed by some process in Π .

No loss If a process decides v at time t , then at least one correct process has received $msgs(v)$ at time t .

The *No loss* property implies that indirect consensus has to be able to know if given v , the messages $msgs(v)$ have been received. This information is provided by the rcv function (the function typically operates on the data structures of the atomic broadcast algorithm that uses indirect consensus).

5.1.4 Reducing atomic broadcast to indirect consensus

The reduction of atomic broadcast to indirect consensus is almost identical to the reduction of atomic broadcast to consensus in [CT96]. The main difference resides in the fact that instead of executing consensus on a set of messages, indirect consensus is executed on a pair (*set of message identifiers, rcv function*). The *Validity* of atomic broadcast is ensured by the *No loss* property of indirect consensus and the *Agreement* property of reliable broadcast: the messages in $msgs(v)$ corresponding to the decision v of indirect consensus are *rdelivered* by at least one correct process (and thus all correct processes eventually *rdeliver* $msgs(v)$).

The atomic broadcast algorithm using indirect consensus is shown in Algorithm 5.1. It shows the following: whenever *abroadcast* is called on a message m , then m is *rbroadcast* to all processes (line 8). If a process *rdelivers* a message that hasn't been ordered yet (line 13), consensus is executed on the unordered message identifiers (lines 15 to 18). The *rcv* function is shown in lines 9 and 10 of Algorithm 5.1.

We now show that the *rcv* function of atomic broadcast satisfies the *Hypothesis A* above. If $rcv(v)$ is true for a correct process, then all messages $msgs(v)$ whose identifier is in v have been previously *rdelivered*. Following the *Agreement* property of reliable broadcast, all correct processes eventually *rdeliver* $msgs(v)$ and thus $rcv(v)$ is eventually true for all correct processes.

5.2 Solving indirect consensus

We now show how to solve indirect consensus. We start by discussing what properties an indirect consensus algorithm must enforce in order to guarantee the *No loss* property. Two consensus algorithms are then adapted into indirect consensus algorithms: (1) the Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm (*CT*) and (2) the Mostéfaoui-Raynal $\diamond\mathcal{S}$ consensus algorithm (*MR*). These two algorithms illustrate two cases. *CT* illustrates the case of a consensus algorithm that is fairly easy to adapt into an indirect consensus algorithm; *MR* illustrates the case of a consensus algorithm whose resilience is reduced by the adaptation into an indirect consensus algorithm (the indirect consensus algorithm requires $\lceil \frac{2n+1}{3} \rceil$ correct processes where the original consensus algorithm required $\lceil \frac{n+1}{2} \rceil$ correct processes).

5.2.1 Conditions on the correctness of indirect consensus algorithms

We present the conditions that indirect consensus algorithms must fulfill in order to ensure the *No loss* property. To do this we introduce the two following definitions:

Definition: v -valent configuration. As in [FLP85], we say that a configuration is *v -valent* at time t if any decision that is taken after t can only be v . As an example, consider a configuration where all processes start consensus at time t_0 with the same initial value v . Such a configuration is *v -valent* at time t_0 (although the first process to decide only does so after t_0).

Definition: v -stable configuration. We say that a configuration is *v -stable* at time t if $f + 1$ processes have received $msgs(v)$ at time t (f is the

Algorithm 5.1: Atomic broadcast algorithm using message identifiers

```

1: Initialisation:
2:    $received_p \leftarrow \emptyset$  {set of messages received by process  $p$ }
3:    $unordered_p \leftarrow \emptyset$  {set of identifiers of messages received but not yet ordered by process  $p$ }
4:   {each message  $m$  has a unique identifier denoted by  $id(m)$ }
5:    $ordered_p \leftarrow \epsilon$  {sequence of identifiers of messages ordered but not yet delivered by  $p$ }
6:    $k \leftarrow 0$  {serial number for consensus executions}

7: procedure  $abroadcast(m)$  {To abroadcast a message  $m$ }
8:    $rbroadcast(m)$  to all

9: procedure  $rcv(ids)$ 
10:  return  $\forall id \in ids : \exists m \in received_p$  such that  $id(m) = id$ 

11: when  $rdeliver(m)$ 
12:   $received_p \leftarrow received_p \cup \{m\}$ 
13:  if  $id(m) \notin ordered_p$  then
14:     $unordered_p \leftarrow unordered_p \cup \{id(m)\}$ 

15: when  $unordered_p \neq \emptyset$  {a consensus is run whenever there are unordered messages}
16:   $k \leftarrow k + 1$ 
17:   $propose(k, unordered_p, rcv)$  { $k$  distinguishes independent consensus executions}
18:  wait until  $decide(k, idSet^k)$ 
19:   $unordered_p \leftarrow unordered_p \setminus idSet^k$ 
20:   $idSeq^k \leftarrow$  elements of  $idSet^k$  in some deterministic order
21:   $ordered_p \leftarrow ordered_p \triangleright idSeq^k$ 

22: {delivers messages ordered and received}
23: when  $ordered_p \neq \emptyset$  and  $\exists m \in received_p$  such that  $head.ordered_p = id(m)$ 
24:   $adeliver(m)$ 
25:   $ordered_p \leftarrow tail.ordered_p$ 

```

maximum number of processes that may crash). v -stability ensures that at least one correct process has received $msgs(v)$.

From these definitions, if a configuration is v -valent or v -stable at time t , any configuration at time $t' > t$ is also v -valent, respectively v -stable.

Ensuring the No loss property We now show that for the *No loss* property to hold, it is necessary and sufficient that any configuration that is v -valent at some time t is also v -stable at t .

We first show that the *No loss* property of an algorithm holds, if the algorithm guarantees that a v -valent configuration is also v -stable. Let us assume that the first decision on a value v is taken at some time t_0 . From the *Uniform agreement* property of the indirect consensus algorithm, the configuration is v -valent at time t_0 . Since the v -valence of a configuration implies that the configuration is also v -stable, v -stability also holds at time t_0 . Thus, the *No loss* property holds.

Now, we show that if an algorithm allows a v -valent configuration that is not v -stable, then the *No loss* property of the algorithm does not hold. Let us assume that the system reaches a v -valent configuration at time t that is not v -stable. Since the configuration is not v -stable, at most f processes have received $msgs(v)$ at time t . If those f processes crash, all copies of $msgs(v)$ are lost and no correct process ever receives $msgs(v)$. Either no decision is taken after time t (and the *Termination* property of the algorithm is violated) or a decision is taken on v , since the system is v -valent at time t . Since $msgs(v)$ are never received by a correct process, the *No loss* property of the algorithm is violated in the latter case.

As a consequence of this result, any indirect consensus algorithm needs to ensure that the relationship “ v -valence $\Rightarrow v$ -stability” for any configuration holds. This relationship between v -valence and v -stability is not trivially satisfied by a consensus algorithm.

5.2.2 Adapting Chandra-Toueg’s $\diamond\mathcal{S}$ consensus algorithm

The following paragraphs present the modification of the *CT* $\diamond\mathcal{S}$ consensus algorithm [CT96] into an indirect consensus algorithm. First of all, a brief overview of the original $\diamond\mathcal{S}$ consensus algorithm is presented, as it helps in understanding the proposed modifications. Then, the necessary modification to that algorithm and v -valence and v -stability are discussed. Finally, the adapted indirect consensus algorithm is presented and proved.

5.2.2.A Chandra-Toueg’s $\diamond\mathcal{S}$ consensus algorithm

The *CT* $\diamond\mathcal{S}$ consensus algorithm proceeds in rounds and requires a majority ($f < \frac{n}{2}$) of correct processes. It behaves as follows: at the beginning of each round, each process sends its estimate of the decision to the process acting as a coordinator in that round. The coordinator waits for a majority of estimates and selects the most recent one (based on its timestamp) and sends it to all processes. At this point, each process either receives the coordinator’s proposal, or suspects the coordinator of having crashed. In the former case, the process sets its own estimate to the coordinator’s proposal, updates its timestamp and sends a positive acknowledgment (*ack*) to the coordinator. In the latter case, a negative acknowledgment (*nack*) is sent. In both cases, the non-coordinator processes proceed to the next round.

The coordinator waits for a majority ($f + 1$) of answers. If all answers are *acks*, the coordinator decides and informs the other processes of its decision. If at least one *nack* is received, the coordinator proceeds to the next round without deciding. It is easy to show that if $\lceil \frac{n+1}{2} \rceil$ of processes have accepted the coordinator’s proposal v , then the system is in a v -valent con-

figuration (*i.e.*, any future decision is v), although the decision on v might only be taken in a later round.

Additional details on the *CT* consensus algorithm are presented in Appendix A.2.1.

5.2.2.B Adapting the algorithm into an indirect consensus algorithm

In the original *CT* algorithm, a process that receives the coordinator's proposal in a given round updates its own estimate to match the proposal of the coordinator (and sends an *ack*). This is precisely the operation that allows the incorrect scenario presented in Section 5.1.2 to occur. Indeed, if all processes blindly adopt the coordinator's proposal v (thus leading to a v -valent configuration, with v a set of message identifiers) and that the originator of $msgs(v)$ crashes, then $msgs(v)$ might be lost and no v -stable configuration of the system can be reached.

In order to avoid this situation, we propose the following modification: whenever a process receives the coordinator's proposal v , it checks if $msgs(v)$ have been received (using the *rcv* function). If so, an *ack* is sent to the coordinator (the proposal is accepted); otherwise, a *nack* is sent (the proposal is refused). Similar approaches have been taken in [CL99, Lam98].

The modified algorithm The pseudo-code of the adapted indirect consensus algorithm is shown in Algorithm 5.2 (the parts that were modified with respect to the original *CT* algorithm have bold line numbers).

The lines 25 to 30 correspond to the modification described above. The *rcv* function is called at line 25 to test if all messages whose identifiers are in the coordinator's proposal have been received. The additional variable $estimate_c$ (lines 2, 18, 20, 21 and 37) represents the coordinator's proposal and can be different from the coordinator's own estimate $estimate_p$ (in case the coordinator does not have the messages corresponding to the estimate v with the highest timestamp). This is explained in the next paragraph.

The need for $estimate_c$ and $estimate_p$ Consider a coordinator c at line 21 that sends v to all in round 1 (c has received $msgs(v)$), and a process p_i that accepts this estimate at line 25 (p_i has received $msgs(v)$). In round 2, the coordinator c' , if it receives the estimate from c or p_i selects it, even if it has not received $msgs(v)$. However, if c and p_i crash later, and no other process has received $msgs(v)$, no correct process might ever receive $msgs(v)$. So, in round 2 the coordinator c' updates $estimate_c$ with v , but $estimate_p$ is still equal to a different value. Once c and p_i crash, the estimate v with timestamp 1 will disappear, and an estimate with timestamp 0 will again be chosen.

 Algorithm 5.2: Chandra-Toueg based $\diamond S$ indirect consensus algorithm
 (code of process p)

```

1: procedure propose( $v_p, rcv$ )
2:    $estimate_p \leftarrow v_p, estimate_c \leftarrow \perp$    { $p$ 's and the coordinator's estimate of the decision value}
3:    $state_p \leftarrow undecided$ 
4:    $r_p \leftarrow 0$                                { $r_p$  is  $p$ 's current round number}
5:    $ts_p \leftarrow 0$                              { $ts_p$  is the last round in which  $p$  updated  $estimate_p$ }
6:   while  $state_p = undecided$  do               {rotate through coordinators until decision reached}
7:      $r_p \leftarrow r_p + 1$ 
8:      $c_p \leftarrow (r_p \bmod n) + 1$            { $c_p$  is the current coordinator}
9:     Phase 1:                                {all processes  $p$  send  $estimate_p$  to the current coordinator}
10:    if  $r_p > 1$  then
11:      send ( $p, r_p, estimate_p, ts_p$ ) to  $c_p$ 
12:    Phase 2:                                {coordinator gathers  $\lceil \frac{n+1}{2} \rceil$  estimates and proposes new estimate}
13:    if  $p = c_p$  then
14:      if  $r_p > 1$  then
15:        wait until [for  $\lceil \frac{n+1}{2} \rceil$  processes  $q$  : received ( $q, r_p, estimate_q, ts_q$ ) from  $q$ ]
16:         $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p \text{ received } (q, r_p, estimate_q, ts_q) \text{ from } q\}$ 
17:         $t \leftarrow$  largest  $ts_q$  such that  $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$ 
18:         $estimate_c \leftarrow$  select one  $estimate_q$  such that  $(q, r_p, estimate_q, t) \in msgs_p[r_p]$ 
19:      else
20:         $estimate_c \leftarrow estimate_p$    {In the first round, the coordinator selects its own estimate}
21:      send ( $p, r_p, estimate_c$ ) to all
22:    Phase 3:                                {all processes wait for new estimate proposed by current coordinator}
23:    wait until [received ( $c_p, r_p, estimate_{c_p}$ ) from  $c_p$  or  $c_p \in \mathcal{D}_p$ ]   {query failure detector  $\mathcal{D}_p$ }
24:    if [received ( $c_p, r_p, estimate_{c_p}$ ) from  $c_p$ ] then   { $p$  received  $estimate_{c_p}$  from  $c_p$ }
25:      if  $rcv(estimate_{c_p})$  then   {check if all messages in  $estimate_{c_p}$  have been received}
26:         $estimate_p \leftarrow estimate_{c_p}$ 
27:         $ts_p \leftarrow r_p$ 
28:        send ( $p, r_p, ack$ ) to  $c_p$ 
29:      else   { $p$  received an estimate  $v$  from the coordinator but  $V$  is missing}
30:        send ( $p, r_p, nack$ ) to  $c_p$ 
31:      else   { $p$  suspects that  $c_p$  crashed}
32:        send ( $p, r_p, nack$ ) to  $c_p$ 
33:    Phase 4:   {the coordinator waits for  $\lceil \frac{n+1}{2} \rceil$  replies. If all replies adopt its estimate, the coordinator rbroadcasts a decide message}
34:    if  $p = c_p$  then
35:      wait until [for  $\lceil \frac{n+1}{2} \rceil$  processes  $q$  : received ( $q, r_p, ack$ ) or for 1 process  $q$ : ( $q, r_p, nack$ )]
36:      if [for  $\lceil \frac{n+1}{2} \rceil$  processes  $q$  : received ( $q, r_p, ack$ )] then
37:        rbroadcast( $p, estimate_c, decide$ ) to all
38:  when  $rdeliver(q, estimate_q, decide)$    {if  $p$  rdelivers a decide message,  $p$  decides accordingly.}
39:    if  $state_p = undecided$  then
40:       $decide(estimate_q)$ 
41:       $state_p \leftarrow decided$ 

```

This scenario illustrates that a process, including the coordinator, only accepts to modify its estimate if it has all the messages corresponding to the identifiers in the new estimate. Since only non-crashed processes send their estimates to the coordinator at the beginning of each round, eventually, only estimates adopted by at least one correct process are received by the coordinator.

5.2.2.C Proof of the algorithm

The *Uniform integrity* and *Uniform validity* properties are trivially proven and are not shown here.

Uniform agreement. The modifications to the algorithm do not affect the *Uniform agreement* property of the original algorithm. Consequently, the proof of the modified algorithm is the same as for the original algorithm by Chandra and Toueg [CT96].

Indeed, the modified algorithm adds a condition on the acceptance of the coordinator's proposal (and the sending of an *ack*). As a consequence, if a decision is taken in the modified algorithm, it would also have been taken in the original algorithm. Thus, since *Uniform agreement* holds in the original algorithm, it also holds in the modified algorithm. \square

Termination. There is a time t such that all faulty processes have crashed. After this time t , all correct processes have an estimate v such that $rcv(v)$ holds. From *Hypothesis A*, there is thus a time t' such that $rcv(v)$ holds for all correct processes and for the estimate v of any correct process. After this time t' , the indirect consensus algorithm behaves exactly like the original consensus algorithm. Thus, if a decision hasn't been taken before t' , the *Termination* property of Chandra-Toueg's $\diamond\mathcal{S}$ consensus algorithm guarantees that the indirect consensus algorithm terminates. \square

No loss. We show that any v -valent configuration is also v -stable. If a configuration is v -valent, it implies that the coordinator always selects v as its proposal. Since the proposal selected by the coordinator is one of the $\lceil \frac{n+1}{2} \rceil$ received estimates, and that the coordinator always receives v at least once, at least a majority of processes have an estimate equal to v .

These $\lceil \frac{n+1}{2} \rceil$ processes that have an estimate equal to v can either (1) have started consensus with v or (2) adopted the proposal v of a previous coordinator, in which case the rcv function on v is verified. In both cases, $msgs(v)$ have been received by a majority of processes and the configuration is v -stable. Since any v -valent configuration is also v -stable, the *No loss* property holds. \square

5.2.3 Adapting Mostéfaoui-Raynal's $\diamond S$ consensus algorithm

We start by presenting an informal overview of the original *MR* consensus algorithm. The problems encountered when adapting this algorithm into an indirect consensus algorithm are then discussed. The solution to these problems, which modifies the resilience of the algorithm, is then presented. Finally, the adapted indirect consensus algorithm is presented and proven correct.

5.2.3.A Mostéfaoui-Raynal's $\diamond S$ consensus algorithm

In [MR99], the authors present a consensus algorithm based on unreliable failure detectors and quorums. We consider their $\diamond S$ based algorithm here. As in the *CT* consensus algorithm, the *MR* algorithm proceeds in rounds and requires a majority ($f < \frac{n}{2}$) of correct processes. In rounds without failures and without suspicions, a decision can be taken within two communication steps by all processes.

Each round consists of two phases. At the beginning of Phase 1, the coordinator of that round sends its estimate to all processes. Each process then either receives the coordinator's proposal, or suspects the coordinator of having crashed. In the latter case, the process considers that an invalid value (\perp) was received from the coordinator. In both cases, the process sends the estimate received from the coordinator (a valid value or \perp) to all processes, which concludes Phase 1 of the algorithm.

In Phase 2, each process waits for a majority of estimates (including the one possibly received from the coordinator). If all received estimates are the same value v , the process decides v and informs all other processes of its decision. If this is not the case, but at least one received estimate is valid (not \perp), the process sets its own estimate to the received valid estimate and proceeds to the next round.

The *Uniform agreement* property of consensus is ensured by the fact that if p decides v , then p has received the estimate v from $\lceil \frac{n+1}{2} \rceil$ processes. This in turn ensures that all processes have received at least one estimate equal to v and have thus set their own estimate to v . Since the estimates of all processes are equal to v , any subsequent decision can only be done on v .

5.2.3.B Problems adapting the Mostéfaoui-Raynal consensus algorithm into an indirect consensus algorithm

As described above, one of the constraints for *Uniform agreement* to hold in the *MR* consensus algorithm is that any process that receives at least one valid estimate must accept that estimate, *i.e.*, modify its own estimate to match the received one. Accepting such an estimate might however lead to

a violation of the *No loss* property of indirect consensus. This is shown by two executions that are indistinguishable for some process p : in one execution the configuration is v -valent but not v -stable; in the other execution the configuration is v -valent and v -stable.

We assume a system with n processes and p a non-coordinator process in the current round of the algorithm. Process p suspects the coordinator and p does not have the messages corresponding to the coordinator's proposal v . The two executions are the following:

(1) the coordinator is correct. $\lceil \frac{n+1}{2} \rceil$ processes accept the proposal of the coordinator, whereas $\lfloor \frac{n-1}{2} \rfloor$ suspect the coordinator. The coordinator receives $\lceil \frac{n+1}{2} \rceil$ estimates equal to its own proposal whereas p receives one estimate equal to the coordinator's proposal and $\lfloor \frac{n-1}{2} \rfloor$ invalid \perp values. In this execution, the coordinator decides. To guarantee the *Uniform agreement* property of consensus, p must accept the coordinator's proposal v (and thus modify its own estimate), although it doesn't have $msgs(v)$.

(2) the coordinator is faulty. Let us assume that the $n - 1$ non-coordinator processes suspect the coordinator and do not have the messages corresponding to the coordinator's proposal v . They thus all send a \perp value. Process p receives the coordinator's proposal as well as $\lfloor \frac{n-1}{2} \rfloor$ invalid \perp values. If the coordinator crashes before any process receives $msgs(v)$, then $msgs(v)$ might be lost. In this execution, p must not accept the coordinator's proposal v .

If p takes a conservative approach and only accepts a proposal v if it has $msgs(v)$ (or that at least one correct process has $msgs(v)$), then the *Uniform agreement* property would be violated in the first execution. If, on the other hand, p takes the optimistic approach of accepting a proposal v even if it doesn't have $msgs(v)$, this could lead to a v -valent configuration that is not v -stable. The *No loss* property of indirect consensus could thus be violated. Therefore, any of the approaches that the algorithm chooses to implement leads to the violation of one of the indirect consensus properties.

The modifications must thus ensure both of the following properties: (i) a process should only accept v if it has $msgs(v)$ or $f + 1$ processes have $msgs(v)$; (ii) if a process decides v in round r , then all non-crashed processes must adopt v during round r . Property (i) and (ii) aim at guaranteeing *No loss*, respectively *Uniform agreement*.

5.2.3.C Modified Mostéfaoui-Raynal algorithm

Consequences on the resilience From the paragraph above, we must modify the *MR* algorithm. In Phase 1 of the algorithm, a process can only accept the coordinator's proposal v if it has received $msgs(v)$ (at this point, a process does not know if any other process has adopted v and can therefore not know if v is stable, which is the second possible condition for

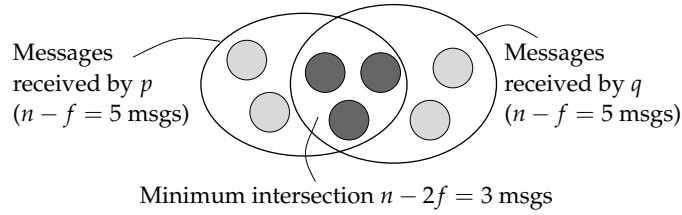


Figure 5.4: Intersection of the estimates received by two processes p and q ($n = 7$ processes and $f = 2$ above).

adopting v). Thus, at the end of Phase 1, when a process p sends the estimate v to all processes, this estimate is a non- \perp value only if p has received $msgs(v)$.

In Phase 2 of the consensus algorithm, all processes wait for $n - f$ estimates from the other processes. If all of these estimates are identical, a decision can be taken. If they are not, a process p can accept a valid estimate v if (1) p has received $msgs(v)$ or (2) if the estimate v was received from at least $f + 1$ processes (*i.e.*, from at least one correct process that has received $msgs(v)$).

To ensure *Uniform agreement*, we have seen that if a decision is taken on v , then all processes must accept v as their own estimate. Not all processes have necessarily received $msgs(v)$ (which means that condition (1) above might not be true for all processes). Therefore, if a decision is taken, the algorithm must ensure that the condition (2) above is true for all processes (*i.e.*, all processes receive at least $f + 1$ estimates equal to v). This can be ensured as follows.

Each process waits for $n - f$ estimates at the beginning of Phase 2. Since there are at most n estimates in the system, each pair of processes receives a common set of estimates. The minimum size of this common set is $n - 2f$ (assuming $f < \frac{n}{2}$), as illustrated in Figure 5.4 (in the case of a system with $n = 7$ processes and at most $f = 2$ failures). So condition (2) is ensured if $n - 2f \geq f + 1$, which leads to $f < \frac{n}{3}$.

The modified MR algorithm The pseudo-code of the adapted *MR* indirect consensus algorithm is shown in Algorithm 5.3 (the parts that were modified with respect to the original algorithm have bold line numbers).

The lines 16 to 19 correspond to the modifications to the first phase of the algorithm. With these modifications, a process accepts the coordinator's proposal $est_from_c_p$ only if it received $msgs(est_from_c_p)$ (in the original consensus algorithm, $est_from_c_p$ is always accepted). In Phase 2, the modifications are two-fold. First of all, the condition $f < \frac{n}{3}$ forces each process to wait for $\lceil \frac{2n+1}{3} \rceil$ estimates at the beginning of Phase 2 (lines 21

 Algorithm 5.3: Mostéfaoui-Raynal based $\diamond\mathcal{S}$ indirect consensus algorithm
 (code of process p)

```

1: procedure propose( $v_p, rcv$ )
2:    $estimate_p \leftarrow v_p$  {estimatep is p's estimate of the decision value}
3:    $state_p \leftarrow undecided$ 
4:    $r_p \leftarrow 0$  {rp is p's current round number}
5:   while  $state_p = undecided$  do {rotate through coordinators until decision reached}
6:      $r_p \leftarrow r_p + 1$ 
7:      $c_p \leftarrow (r_p \bmod n) + 1$  {cp is the current coordinator}
8:      $est\_from\_c_p \leftarrow \perp$  {estimate received from the coordinator or invalid ( $\perp$ )}
9:     Phase 1: {coordinator proposes new estimate; other processes wait for this new estimate}
10:    if  $p = c_p$  then
11:       $est\_from\_c_p \leftarrow estimate_p$ 
12:      send ( $p, r_p, est\_from\_c_p$ ) to all
13:    else
14:      wait until [received ( $c_p, r_p, est\_from\_c_p$ ) from  $c_p$  or  $c_p \in \mathcal{D}_p$ ] {query fail. det.  $\mathcal{D}_p$ }
15:      if received ( $c_p, r_p, est\_from\_c_p$ ) from  $c_p$  then {p received  $est\_from\_c_p$  from  $c_p$ }
16:        if  $rcv(est\_from\_c_p)$  then
17:           $est\_from\_c_p \leftarrow est\_from\_c_p$ 
18:        else
19:           $est\_from\_c_p \leftarrow \perp$ 
20:        send ( $p, r_p, est\_from\_c_p$ ) to all
21:      Phase 2: {each process waits for  $\lceil \frac{2n+1}{3} \rceil$  replies. If they indicate that  $\lceil \frac{2n+1}{3} \rceil$  processes adopted the proposal, the process rbroadcasts a decide message}
22:      wait until for  $\lceil \frac{2n+1}{3} \rceil$  processes  $q$  : received ( $q, r_p, est\_from\_c_q$ )
23:       $rec_p \leftarrow \{est\_from\_c_q \mid p \text{ received } (q, r_p, est\_from\_c_q) \text{ from } q\}$ 
24:      if  $rec_p = \{v\}$  then
25:         $estimate_p \leftarrow v$ 
26:        call takedecision
27:      else if  $rec_p = \{v, \perp\}$  then {accept v if  $rcv(v)$  is true or v received  $\lceil \frac{n+1}{3} \rceil$  times}
28:        if  $rcv(v)$  or p received ( $-, r_p, v$ ) from more than  $\lceil \frac{n+1}{3} \rceil$  processes then
29:           $estimate_p \leftarrow v$ 
30: procedure takedecision
31:   decide( $estimate_p$ )
32:    $state_p \leftarrow decided$ 
33:   send ( $p, estimate_p, decide$ ) to all {ad-hoc reliable broadcast}
34: when receive ( $q, estimate_q, decide$ ) {if p receives a decide message, p decides accordingly}
35:   if  $state_p = undecided$  then
36:      $estimate_p \leftarrow estimate_q$ 
37:     call takedecision

```

and 22). Secondly, if a process receives a valid estimate v as well as \perp values, the valid estimate is adopted if (1) the process has $msgs(v)$ or (2) if the estimate v was received from more than one third of the processes (lines 28 and 29).

The remaining parts of the indirect consensus algorithm are identical to the original *MR* consensus algorithm.

5.2.3.D Proof of the algorithm

The *Uniform integrity* and *Uniform validity* properties are trivially proven and are therefore not shown here.

Uniform agreement. Let process p be the first process that sends a decision message and then decides on some value v . Process p previously received the estimate v from $\lceil \frac{2n+1}{3} \rceil$ processes in Phase 2 of a given round r . All other processes also received $\lceil \frac{2n+1}{3} \rceil$ values in round r and thus received at least $\lceil \frac{n+1}{3} \rceil$ identical values to p . Thus, all processes eventually execute line 25 or 29 in round r and set their own estimate to v . After round r , the estimate of all processes is thus equal to v . \square

Termination. The proof of *Termination* is similar to the case of the Chandra-Toueg based indirect consensus algorithm. There is a time t such that all faulty processes have crashed. After this time t , all correct processes have an estimate v such that $rcv(v)$ holds for at least one correct process. From *Hypothesis A*, there is thus a time t' such that $rcv(v)$ holds for all correct processes and for the estimate v of any correct process. After this time t' , the indirect consensus algorithm behaves exactly like the original consensus algorithm (albeit with the different level of resilience). Thus, if a decision hasn't been taken before t' , the *Termination* property of Mostéfaoui-Raynal's $\diamond\mathcal{S}$ consensus algorithm guarantees that the indirect consensus algorithm also terminates. \square

No loss. In the modified Mostéfaoui-Raynal indirect consensus algorithm, a system is in a v -valent configuration if $\lceil \frac{2n+1}{3} \rceil$ processes accept the same estimate v in a given round r . The estimate of a process is equal to v in three cases : (1) consensus was executed with v as the initial proposal, (2) the process received v in Phase 1 or 2 and accepted it because $msgs(v)$ had been previously received or (3) the process received v in Phase 2 from at least $f + 1$ processes. Since at least $\lceil \frac{2n+1}{3} \rceil$ processes have the same estimate v in the v -valent configuration, at least $f + 1$ processes must have modified their estimate following cases (1) or (2) above. In both of these cases, the processes have $msgs(v)$. The configuration is thus v -stable, since at least $f + 1$ processes have $msgs(v)$. Since any v -valent configuration is also v -stable, the *No loss* property is verified. \square

5.3 Performance measurements

In Section 5.1, we presented a performance comparison between atomic broadcast with consensus on messages and consensus on message identifiers. In the following paragraphs, we present measurements comparing atomic broadcast using reliable broadcast and indirect consensus to (the faulty implementation of) atomic broadcast using reliable broadcast and consensus directly on message identifiers, in order to estimate the overhead introduced by the indirect consensus solution.

We also present a comparison between atomic broadcast using indirect consensus to (the correct implementation of) atomic broadcast using *uniform* reliable broadcast and consensus directly on message identifiers.

This section starts by a presentation of the system setup and the Neko framework that was used in the experiments. Several comparisons between indirect consensus and the faulty and correct implementations using consensus are then presented.

5.3.1 System setup and the Neko framework

The benchmarks were run on two setups:

Setup 1 is a cluster of PCs running Red Hat Linux (kernel 2.4.18). The PCs have Pentium III 766 MHz processors and 128 MB of RAM, and are interconnected by a 100 Base-TX Ethernet. The Java Virtual Machine was Sun's JDK 1.4.1_01.

Setup 2 is a cluster of machines running SuSE Linux (kernel 2.6.11). The machines have Pentium 4 processors at 3.2 GHz and have 1 GB of RAM. They are interconnected by Gigabit Ethernet and run Sun's 1.5.0 Java Virtual Machine.

The protocols are implemented as layers of a stack in the Neko framework [UDS02]. The *CT* atomic broadcast algorithm was implemented and executed either on messages or on message identifiers, according to the test configuration. The indirect consensus algorithm was implemented based on an already existing implementation of the *CT* $\diamond S$ consensus algorithm that was used in previous performance studies [USS03, UHSK04] and for the simulations in Section 4.1.6. All the results presented here were obtained on the real networks described above.

5.3.2 Performance metric: latency versus throughput and message size

The performance metric for atomic broadcast is the *latency*, defined as the average (over all processes) of the elapsed time between *abroadcasting* a

5.3. Performance measurements

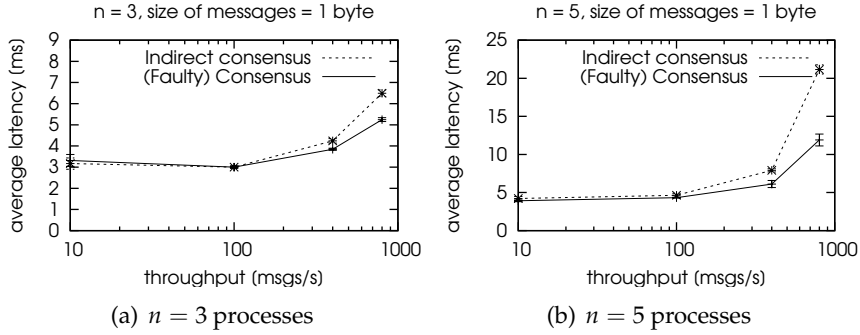


Figure 5.5: Latency vs. throughput of the atomic broadcast algorithm using indirect consensus or (faulty) consensus on message identifiers in a system with 3 and 5 processes, in *Setup 1*.

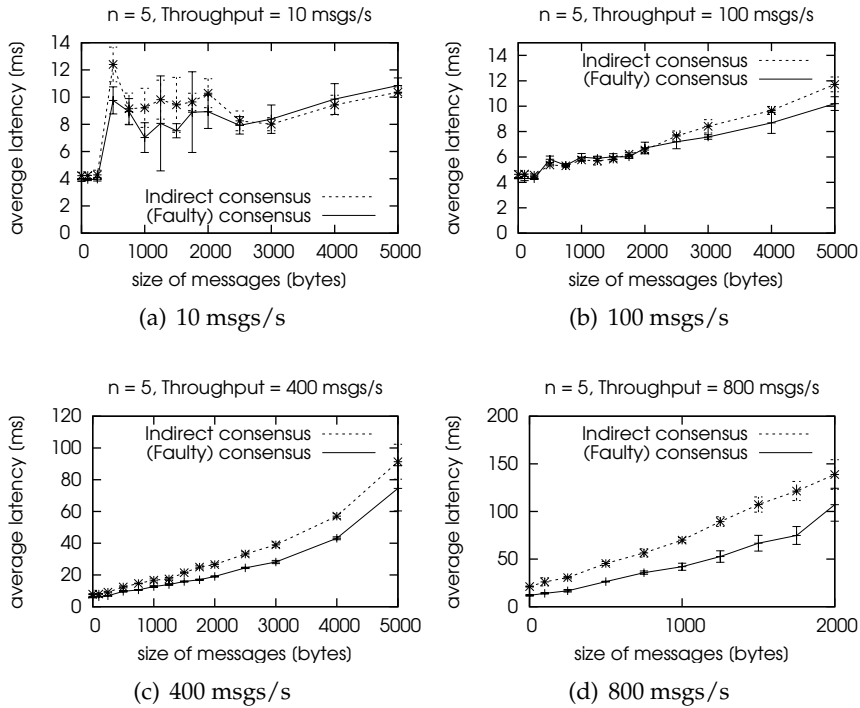


Figure 5.6: Latency vs. payload of the atomic broadcast algorithm using indirect consensus or (faulty) consensus on message identifiers in a system with 5 processes, in *Setup 1*.

message m and *adelivering* m . A simple symmetric workload is used: all processes *abroadcast* messages at the same rate and the global rate is called the *throughput*.

To quantify and compare the performance of indirect consensus with the faulty and correct implementations of atomic broadcast using consensus directly on message identifiers, we present figures showing the latency of atomic broadcast as a function of the message size (for low and high throughputs) and as a function of the throughput (for a given message size).

5.3.3 Performance results: overhead of indirect consensus

Overhead as a function of the throughput Figure 5.5 presents the performance comparison between indirect consensus and the faulty implementation of consensus directly on message identifiers. The size of the messages is set to one byte. Figure 5.5 shows that the overhead of indirect consensus increases as the throughput increases in a system with 3 or 5 processes in *Setup 1*.

This result is not surprising: as the throughput increases, consensus is done on larger sets of messages. The calls to the *rcv* function (to verify if the messages whose identifiers are in a consensus proposal have been received) thus take more and more time. When the throughput is low (10 or 100 messages per second), the overhead of indirect consensus is negligible. As the system throughput increases, the overhead also increases and reaches at most 1.3 ms in a system with 3 processes and 9.5 ms in a system with 5 processes. This is the price to pay for a correct implementation, since executing an unmodified consensus algorithm directly on message identifiers can lead to a violation of atomic broadcast's validity property in a system where one process fails.

Overhead as a function of the payload Figure 5.6 compares the performance of indirect consensus and consensus directly on message identifiers as the size of the messages increases (in *Setup 1*). The overhead ratio remains stable as the size of the messages varies. In the case of low throughputs (10 messages per second), the overhead is negligible for all message sizes. For higher throughputs, the overhead is clearly measurable, but does not vary much as the size of the messages increases. These results are expected: since both algorithms only use the message identifiers to reach a decision, the messages themselves (and thus their size) do not affect the performance of the indirect consensus and consensus algorithms.

5.3.4 Performance results: comparison of two correct approaches

Overhead as a function of the payload Figures 5.7 and 5.8 presents the latency of atomic broadcast as a function of the payload of the messages in a system with 3 processes in *Setup 2*. Atomic broadcast uses either (i) reliable broadcast and indirect consensus or (ii) *uniform* reliable consensus and consensus directly on message identifiers. Both solutions are correct. The uniform reliable broadcast algorithm that we consider supports up to $f < \frac{n}{2}$ crash-failures and requires $O(n^2)$ messages and 2 communication steps to deliver a message that was previously broadcast.

Figure 5.7 shows that if reliable broadcast needs $O(n^2)$ messages (as in [CT96]), then indirect consensus and reliable broadcast achieve slightly lower latencies than consensus on message identifiers and uniform reliable broadcast. This slight difference is attributed to the additional communication step (and message processing) that uniform reliable broadcast needs, compared to reliable broadcast.

Figure 5.8 shows that if reliable broadcast only needs $O(n)$ messages in good runs (when using a failure detector for example), the performance of indirect consensus is clearly better than if consensus and uniform reliable broadcast are used to solve atomic broadcast.

Overhead as a function of the throughput Figure 5.9 presents the latency of the atomic broadcast algorithm (using either indirect consensus and reliable broadcast or consensus and uniform reliable broadcast) as a function of the throughput in *Setup 2*. The payload of all messages is one byte. The figure shows that the performance of atomic broadcast using consensus and uniform reliable broadcast degrades significantly as the throughput increases. Atomic broadcast using indirect consensus and reliable broadcast behaves similarly (although slightly better) if reliable broadcast needs $O(n^2)$ messages (Figure 5.9(a)). If a reliable broadcast algorithm requiring $O(n)$ messages is used however, then the performance of atomic broadcast using indirect consensus is much less affected by the throughput than before (Figure 5.9(b)).

5.3.5 Overview of the performance results

In Section 5.1, we saw that executing atomic broadcast using indirect consensus (on message identifiers) provides better performance than using consensus on messages, especially as the sizes of the system or the messages increase. In previous group communication stack implementations, consensus was often executed directly on message identifiers, which can lead to faulty executions if a process crashes. The indirect consensus ap-

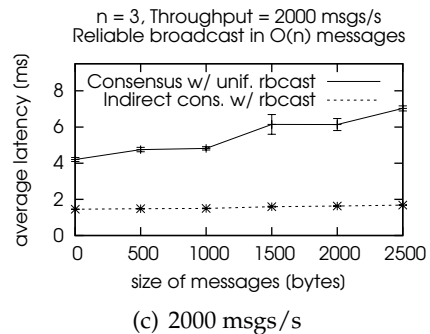
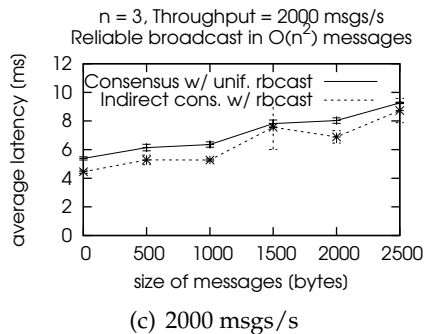
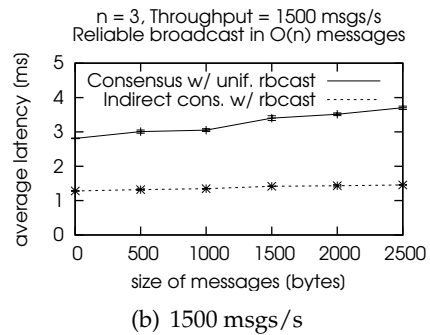
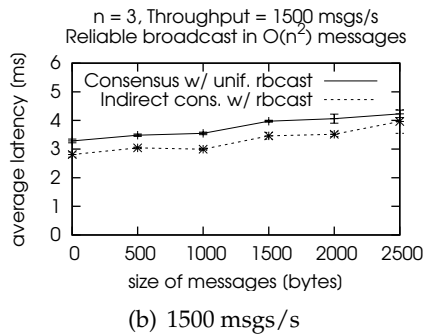
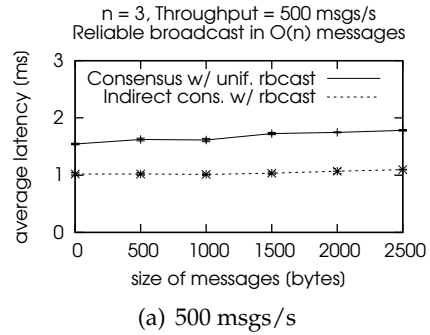
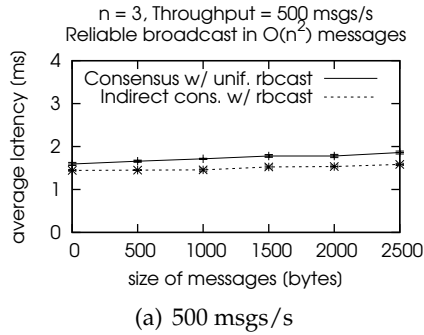


Figure 5.7: Latency vs. payload of the atomic broadcast algorithm using indirect consensus and reliable broadcast or consensus on message identifiers and uniform reliable broadcast in a system with 3 processes on *Setup 2*. The reliable broadcast algorithm uses $O(n^2)$ messages for each *rbroadcast*.

Figure 5.8: Latency vs. payload of the atomic broadcast algorithm using indirect consensus and reliable broadcast or consensus on message identifiers and uniform reliable broadcast in a system with 3 processes on *Setup 2*. The reliable broadcast algorithm uses $O(n)$ messages for each *rbroadcast*.

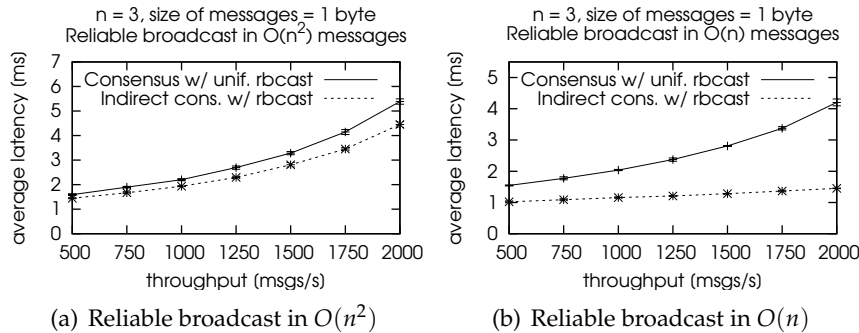


Figure 5.9: Latency vs. throughput of the atomic broadcast algorithm using indirect consensus and reliable broadcast or consensus on message identifiers and uniform reliable broadcast in a system with 3 processes on *Setup 2*. The reliable broadcast algorithm uses either $O(n^2)$ or $O(n)$ messages for each *rbroadcast*. The payload of each message is 1 byte.

proach, which solves this problem, yields performance results that are comparable to the faulty solution (in the case of an indirect consensus algorithm with the same degree of resilience as the corresponding consensus algorithm), as Figures 5.5 and 5.6 show.

Furthermore, compared to the other correct solution (atomic broadcast using *uniform* reliable broadcast and consensus directly on message identifiers), the combination of atomic broadcast and indirect consensus achieves better performance (especially if one considers a reliable broadcast algorithm that only needs $O(n)$ messages per *rbroadcast*), as illustrated by Figures 5.7, 5.8 and 5.9.

The cost of adopting a correct implementation of atomic broadcast on message identifiers is thus fairly low and ensures that the properties of atomic broadcast hold, even if processes crash.

5.4 Discussion

In [CT96], atomic broadcast is reduced to consensus on messages. This reduction is correct, but since consensus is executed on sets of messages, it yields poor performance as the size of the messages increases. Instead, consensus can be executed on message identifiers, which decouples the consensus algorithm from the size of the messages. This can however lead to the violation of the *Validity* property of atomic broadcast. *Indirect* consensus addresses this issue by providing a *No loss* property, which guarantees that all messages whose identifiers have been decided upon are eventually

delivered by atomic broadcast. To ensure the *No loss* property, the indirect consensus algorithm must guarantee that any v -valent configuration (any future decision is v) is also v -stable (at least one correct process has received the messages whose identifiers are in v).

This chapter has shown that adapting a consensus algorithm into an indirect consensus algorithm is not trivial. The resilience of the adapted Mostéfaoui-Raynal $\diamond\mathcal{S}$ indirect consensus algorithm is $f < \frac{n}{3}$ whereas the original consensus algorithm supports $f < \frac{n}{2}$ failures. Chandra-Toueg's $\diamond\mathcal{S}$ -based consensus algorithm does not have this problem and was easy to adapt.

Finally, the performance of the Chandra-Toueg based indirect consensus algorithm is better than the original consensus algorithm on messages and comparable to the performance of the faulty implementation of the consensus algorithm directly on message identifiers. The performance of the implementation using indirect consensus algorithm is also better than the performance of atomic broadcast using uniform reliable broadcast and consensus.

Part II

Experimental Evaluation of Atomic Broadcast Algorithms

Robust TCP connections for fault tolerant computing

When processes on two different machines communicate, they most often do so using the TCP protocol [Ste94]. The reasons for the popularity of TCP are threefold. Firstly, it offers a convenient interface to communication: a bi-directional byte stream. Secondly, it hides most problems of the communication channel from the programmer: message losses, duplicates and short losses of connectivity. Thirdly, it is extremely flexible and well engineered: it suits needs as different as short lived HTTP sessions, long lived file transfers, and continuous low traffic sessions like a remote login. Moreover, TCP can work on low-latency reliable local networks and on the high-latency not-so-reliable Internet with acceptable performance.

While TCP is appropriate for a wide range of applications, it has shortcomings in other application areas. One of these areas is fault-tolerant distributed computing. Many algorithms in fault-tolerant distributed computing assume so called *quasi-reliable channels*: if a process p sends a message m to process q , m will eventually be received by q if neither p nor q fails [BCBT96]. An obvious way to implement quasi-reliable channels is to use a TCP connection between p and q . Unfortunately, TCP does not address link failures adequately: TCP breaks the connection if connectivity is lost for some duration (typically minutes, but the connection is only broken if TCP actually wants to send data or if keepalives are sent). This might sometimes be undesirable, and hence we need a way to recover from broken TCP connections. Potential applications that would benefit from such a feature are all applications that are willing to wait for connectivity longer than the default TCP parameters allow. Examples include long-lived remote login sessions on a computer not permanently connected to the Internet (mobile devices or a PC with a modem). There are more elaborate examples from fault tolerant distributed computing. The explanation requires some additional context.

In fault-tolerant distributed computing, process failures and link fail-

ures are often abstracted using *group membership*. A group membership service offers each process a *view* of the system, the set of processes the process can currently communicate with. The view changes over time as (1) processes crashes and recover, or (2) link failures occur and are repaired [CBDS02]. There are two kinds of group membership: (1) *primary partition* group membership, in which processes agree on the sequence of views, and (2) *partitionable* group membership in which multiple concurrent views can simultaneously exist. In each case, broken TCP connections can be used to trigger changes in views [CBDS02]. We shall argue here that this is not a good idea when using primary partition group membership; consequently, link failures should be transparent, and we can achieve this by robust TCP connections. Consider a replicated server with three replicas s_1, s_2, s_3 . Assume a partition failure which partitions s_3 away from s_1 and s_2 . If link/partition failures are transparent, nothing needs to be done when the partition failure is repaired. In contrast, if failures are not transparent, all server updates that took place during the partition failure need to be explicitly forwarded to s_3 (by s_1 or s_2). A detailed discussion of this issue can be found in [CKV01].

Robust TCP connections present a solution to the problem of broken TCP connections. Robust TCP connections have the same interface and properties as standard TCP connections, except that these connections never break due to network problems (and thus implement the quasi-reliable channel abstraction). We define a session layer protocol on top of TCP that ensures reconnection, and provides exactly-once delivery for all transmitted data. A prototype has been implemented as a Java library (however, nothing prevents a C implementation). The prototype has less than 10% overhead on TCP sockets with respect to the most important performance characteristics: response time and throughput. Robust TCP sockets integrate seamlessly into Java. Source code integration is done by replacing occurrences of `new Socket` and `new ServerSocket` by calls that create instances of our replacement classes. Binary integration requires a few changes in the Java core libraries: these changes would make it possible to replace sockets with robust sockets in a Java application without re-compilation and without changing the application.

The rest of the chapter is structured as follows. Section 6.1 discusses design issues for robust TCP connections. Section 6.2 presents the protocol. Section 6.3 discusses the implementation of robust TCP connections in Java. Performance figures are given in Section 6.4. Related work is discussed in Section 6.5, and Section 6.6 concludes the chapter.

6.1 Design of the protocol

6.1.1 Requirements

In this section, we present our requirements for robust TCP connections, along with their implications for the design of the protocol.

Using robust TCP should be as transparent for the user as possible. Therefore robust TCP should have the same interface as standard TCP connections and offer the same service: a bidirectional stream of bytes. For our prototype, this implies that robust TCP should offer the Java sockets interface (integration into Java is discussed in detail in Section 6.3.2). Our functional and non-functional requirements are the following:

Duration of connection. In standard TCP connections, the connection is closed whenever data is sent on the connection, but no acknowledgment is received for several minutes. Robust TCP connections should only be closed if the application explicitly requests this. This means that robust TCP does not have any timeout mechanism that might lead to break the connection. Specifically, robust TCP connections must survive link failures and network partitions.

Flow/congestion control. Robust TCP should have the same flow control / congestion control mechanisms and behavior as TCP. It should use buffers of limited size.

Performance. Robust TCP should incur an overhead of less than 10% on normal operation, with respect to all relevant performance figures: response time, throughput. The overhead on the time to open/close connections is less important, as robust TCP connections are long-lived. Also, the overhead on the network should be a small fraction of the overall traffic.

Easy deployment. The implementation should be lightweight and deployment should be easy. Robust TCP will require extensions at both endpoints of the connection, as losses of connectivity affect both sides: modifying just one endpoint is not sufficient. However, we do not want to rely on daemons supporting our protocol. Moreover, we want a user space implementation, with no need to modify the kernel or to have administrator privileges.

The easy deployment requirement implies that we can modify neither TCP nor any of the lower layers, nor can we configure the parameters of these layers.

Modifications to the TCP kernel code would be very small and would essentially mean changing a timeout value from some number of minutes

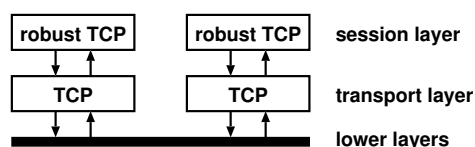


Figure 6.1: The robust TCP protocol in the OSI reference model.

to $timeout = \infty$. This would guarantee that TCP never closes connections due to a timeout.

However, modifying TCP has several disadvantages: first of all, it would require modifications to kernel code. This would of course reduce the portability of the code since many different TCP stack implementations exist and would all need to be modified. For non-open-source platforms, the integration of robust TCP would be complicated, if possible at all.

Secondly, the user of the protocol would have to re-compile the kernel in order to be able to use robust TCP. The average user is not necessarily comfortable with this and most users will not blindly trust new kernel code (an unintentional or malicious misbehavior can never be excluded).

For these reasons, we decided to implement a protocol on top of TCP, in the session layer of the OSI reference model (Figure 6.1). The purpose of the protocol is to deal with broken TCP connections. Most performance requirements are easily fulfilled if the overhead of the session protocol is always just a small fraction of the traffic generated by TCP. We discuss performance issues in more detail in Section 6.4.

In addition to the standard TCP interface, the application might want to be informed about the state of session connections (robust TCP connections do not need to be the same black box to the application as standard TCP connections). We plan to extend the interface to provide the following information: the number of bytes sent but not acknowledged, the time elapsed since the last send operation whose data was not acknowledged, and the duration for which a receive (or send) operation has been blocked.

6.1.2 Issues at the session layer

TCP will close a connection if two hosts cannot contact each other for several minutes and data is being exchanged. When this happens, the session protocol (1) must reconnect the two parties, (2) must be able to uniquely identify a connection, and (3) must ensure that all data sent is received exactly once.

The issues related to reconnection are the following. First of all, the client (the party that did an active open) must initiate the reconnection to the server (the party that did a passive open), and not vice-versa. The reason is simply that only the server has a static address to connect to (further-

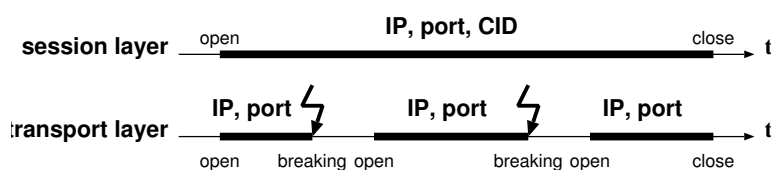


Figure 6.2: Lifetime of session and transport layer connections.

more, server to client connections are problematic if the communication parties are separated by firewalls). This implies that the server cannot close the socket on which it listens for TCP connections when it is no longer willing to accept new connections. This socket needs to remain open as long as there are active session layer connections. The second issue is that the reconnection attempt might fail. In this case, the client should repeatedly try reconnecting.

A session layer connection is potentially associated with multiple transport layer connections (Figure 6.2). This means that we need to identify the session layer connection upon reopening a TCP connection. A session layer connection is uniquely identified by the combination of (1) the IP address and the port number of the TCP socket on the server side, along with (2) a unique connection identifier (CID) generated by the robust TCP server (session layer) upon the first connection attempt. This also allows us to distinguish a reconnection attempt from the first connection attempt of a new session layer connection.

When a TCP connection is broken, we do not know how much data has been successfully transmitted (we cannot access the information in TCP acknowledgments). Therefore all transmitted data must be buffered and retransmitted upon reconnection if necessary. As the protocol should only use a buffer of limited size, it has to exchange *control messages* to acknowledge received data. Upon reception of such a message, a part of the buffer can be discarded. We must make sure that the acknowledgments constitute only a small portion of the overall traffic generated by the connection. Also, flow control issues arise if the buffer fills up.

6.1.3 The problem of control messages

The control messages can be passed between the client and the server in two ways: either the messages are passed in-band, multiplexed with application data, or out-of-band, on a different channel. We have chosen an out-of-band solution, primarily because the in-band solution poses severe performance problems, and secondarily because it is more complex. To understand why, let us first explain how an out-of-band solution could be implemented. The idea is simple: the session layer *send* operation buffers

outgoing data, and the session layer *receive* operation sends acknowledgments. Also, a lightweight flow control mechanism is needed that blocks the *send* operation as long as the outgoing data buffer is full.

Let us contrast this implementation to the in-band solution. The problems are the following:

- Multiplexing and demultiplexing two streams may be costly in itself, especially if data is transmitted in small chunks. This is the easiest problem: a solution similar to Nagle's algorithm [Nag84] could offer acceptable performance.
- The data stream and the stream of control messages are independent: even if no data is sent to one of the communication parties, that party may still receive control messages. For this reason, each party has to constantly read the TCP stream to check for control messages. This requires a dedicated *control* thread to read the socket. So, when data arrives, it has to pass through the control thread before reaching the application thread that reads the socket. This leads to context switching and one extra copy of the data to an intermediate buffer, and yields poor performance.
- The solution requires a rather complex flow control mechanism. If the control thread receives a lot of data and the application is not ready to receive data, the intermediate buffer fills up. The control thread must continue reading, in order not to miss control messages. This implies that the protocol has to discard any further data, and has to ask the other side for retransmission. In contrast, an out-of-band solution only needs retransmission of data when the TCP connection breaks.

The question remains how to pass out-of-band control messages. The two choices are (1) UDP datagrams and (2) a separate TCP connection. We chose the UDP solution for reasons of performance and resource utilization. Indeed, the TCP solution needs twice as many TCP connections and TCP ports on each side (2 per session layer connection). In contrast, a server can share the UDP control port among all the connections it manages. Also, the TCP solution exchanges more IP packets during the whole lifecycle of the connection (open, data transfer and close).

The UDP solution might seem more complex at first: (1) we need to identify the connection in each control message, and (2) we need to ensure reliable delivery and FIFO order of control messages (we can afford losing some acknowledgments, though). However, the TCP solution would result in an equally complex implementation, as it would have to ensure reliable delivery and FIFO order as well (in case the TCP connection for control messages breaks).

Finally, note that the TCP solution should be preferred if the connection passes through a firewall, as firewalls are usually configured to reject UDP

packets. However, this was not a problem for us, and if the need arises, the protocol can be easily modified to use TCP.

6.2 The session layer protocol

A robust TCP session has three phases: (1) connection establishment (opening), (2) data exchange, and (3) connection termination (closing). Whenever TCP errors occur, the protocol enters the reconnection phase. We now describe each of these phases in detail, and then discuss how TCP errors are handled.

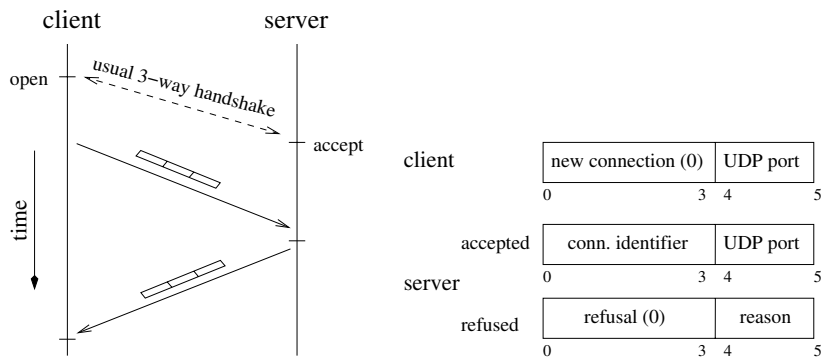


Figure 6.3: Opening phase of robust TCP

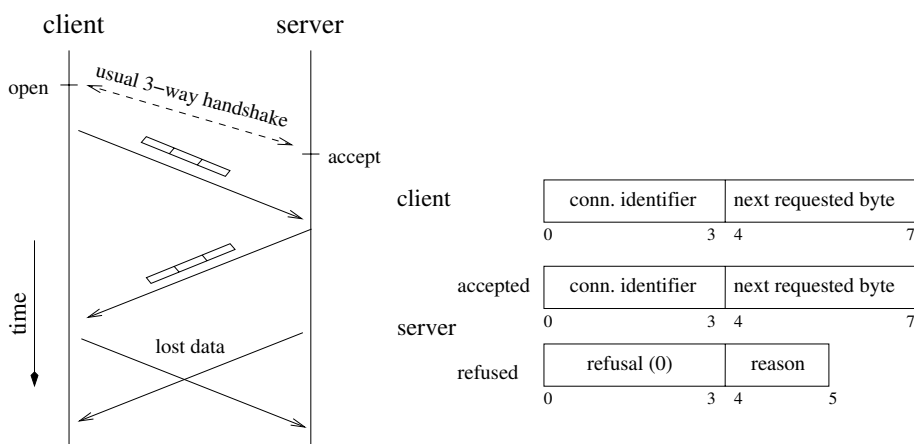


Figure 6.4: Reconnection phase of robust TCP

6.2.1 Opening a connection and reconnection

The client starts by establishing a TCP connection with the server, sends (on the TCP connection) the *new connection* control message together with the number of the UDP port used for exchanging control messages (Figure 6.3), and then waits for a unique connection identifier (CID) from the server. The server assigns CIDs in the order $k, k + 1, k + 2$, etc. k is chosen randomly when the server starts up, in order to avoid that clients of a server that used to listen on the same port (and that quit or crashed) confuse the server (one faces a similar issue when choosing initial TCP sequence numbers).

The CID is always chosen to be different from 0, in order to give a special meaning to 0: it is used to distinguish a new session from an existing session that re-opens its TCP connection. If the server accepts the connection, a new unique session identifier is sent to the client, together with the server UDP control port. The details of the protocol are shown in Figure 6.3.

New robust TCP sessions can be refused. This happens when the server has closed the session layer socket, or when it temporarily runs out of resources (too many open connections). In that case, the server sends a *refusal* (0) control message (instead of a non-null connection identifier) and a reason code. The client will try to reconnect if the reason code indicates a temporary reason for refusal.

If the TCP connection breaks during data exchange, re-establishing the TCP connection uses a similar protocol, shown in Figure 6.4. The client opens a new TCP connection to the server, then sends the session layer connection identifier and the number of bytes received on this session (we use a 32 bit counter that wraps around to 0 when it reaches 2^{32}). The server answers by re-sending the connection identifier to confirm that the re-connection has been accepted and the number of bytes received. As soon as any of the parties knows how many bytes have been received by the other party, it starts retransmitting data that was lost. Finally, the session is ready again for exchange of data. To the user, the session appears to be open during the whole reconnection process, *i.e.*, send and receive calls return normally.

6.2.2 Data exchange

Anytime the user writes some data on the robust TCP connection, the data is stored in a buffer and then sent on the TCP connection. The protocol also maintains a counter for the total number of bytes sent on the robust TCP connection. Whenever an acknowledgment is received, acknowledged data is removed from the buffer. The acknowledgment, similarly to TCP's acknowledgments, points to the next byte that the reader process is expecting to receive. The exact format of acknowledgments and other control messages is discussed later.

The buffer for outgoing data is of limited size, hence some flow control is needed. When the buffer fills up, the protocol blocks send operations. Also, it sends a control message that forces the other party to acknowledge data. We set the default parameters such that send operations hardly ever block: acks are sent after every 8 kilobytes received and the outgoing buffer is of size 16 kilobytes (these parameters can be configured by the user to suit its execution environment). Also note that the default setting yields few acknowledgment packets compared to the number of data packets.

6.2.3 Closing the connection

We only discuss how the session is closed; handling of half-close is analogous. Closing the session is done by closing the TCP connection; control is returned immediately to the application. However, the two sides have to notify each other that the connection is closed, in order to free up resources associated to the connection. This notification is done asynchronously, using control messages. Thus, the UDP control ports are active until these notifications have been exchanged.

If the server closes the session and the client still wants to use it, TCP generates an error. It is possible that the server could not notify the client about the closing of the session at this point. In this case, the client will try to reconnect, and the server implicitly notifies the client about the closing of the session by rejecting the reconnection attempt.

6.2.4 Handling TCP errors

Whenever a TCP socket operation returns an error, the protocol first tries to gracefully close the TCP socket. Then, if the error occurred during the opening phase, the opening phase is restarted. Otherwise, the protocol enters the reconnection phase — or re-enters the reconnection phase if the error occurred in the reconnection phase.

The protocol allows to increase the delay between two consecutive reconnection attempts, using an exponential backoff strategy. Furthermore, some TCP errors indicate the failure of the other party rather than the loss of connectivity [NF97]. We could use this information to avoid unnecessary reconnection attempts.

6.2.5 UDP control messages

The structure of the UDP control messages is presented in Figure 6.5. The message starts with eight header bytes. The first four bytes of this header store the identifier of the connection. The next byte defines the type of the message: ACK to acknowledge data, REQ_ACK to force the other side to

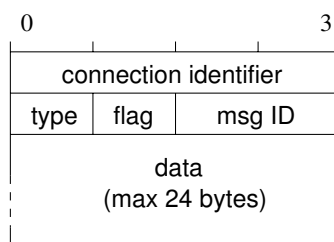


Figure 6.5: Structure of robust TCP control messages.

send an acknowledgment, and CLOSE, HALF_CLOSE and RESET to manage closing connections. Since UDP does not guarantee delivery, control messages that need to be acknowledged (such as a CLOSE message initiating a three-way handshake) are flagged using the flag byte. The header ends with a two byte identifier of the control message. These two bytes have a meaning only if the control message needs to be acknowledged, as specified by the flag byte. An acknowledgment control message (type CONF) will carry this identifier to indicate that the message has been received. Finally, the message ends with a data section. For acknowledging data (ACK and REQ_ACK) four bytes represent the number of bytes received. The other control messages have no data section.

Note that we could have compressed the data in control messages. However, this is not worthwhile as the overhead is negligible compared to the overhead of UDP, IP and the lower layers. In particular, control messages easily fit into the smallest Ethernet frame.

6.3 Java Implementation

6.3.1 Classes

The implementation of robust TCP provides the same interface as the sockets in `java.net`. For a smooth integration into existing programs, the classes implementing robust TCP sockets, *i.e.*, `RSocket` (client side) and `RServerSocket` (server side), extend the classes `Socket` and `ServerSocket` of the standard Java TCP interface. The slight differences between the client and server side of a connection (in the reconnection procedure) are handled by a class that extends `RSocket` and implements the server side specificities (*e.g.*, the reconnection procedure and the notifications that complete a close). This class is package-private and is instantiated only by the `RServerSocket` whenever a new connection is created.

Both endpoints also each need a dedicated thread that constantly reads control messages from the UDP socket. All connections in the same JVM (both client and server side) share this UDP socket. This means that a single

control thread is used to read the control messages and dispatch them to the right connection. Using few threads is essential for achieving good performance on the server side.

6.3.2 Integration into Java

Since the robust TCP sockets extend the Java sockets, the user can simply replace any call to the constructor of `Socket` or `ServerSocket` by a similar call to `RSocket` and `RServerSocket`. This is a rather easy way to integrate robust TCP connections into new applications or code available in source code form.

Let us now discuss how to integrate robust TCP connections into existing applications without changing the source code. Java provides a way to use modified sockets instead of the standard ones without modifying or re-compiling the application. The user of the Java libraries can call the method `ServerSocket.setSocketFactory` (for the server side) and `Socket.setSocketImplFactory` (for the client side) with as parameter an object that will serve as a factory for socket implementations. Socket implementations extend the `SocketImpl` class. Similarly to C sockets, this class provides a client interface to connect to a remote host (`bind` and `connect`), and a server interface to accept connections (`bind`, `listen` and `accept`).

Unfortunately, Java socket factories are not flexible enough to allow the integration of robust TCP sockets (that is, without modifying the Java core libraries). The methods `setSocketFactory` and `setSocketImplFactory` can only be called once in an application and no plain Java sockets can be created after the call. This is a problem for us, as we access TCP by plain Java sockets. However, several requests are present in the Java bug tracking database [Soc99] that aim at making the socket factory features more flexible. Once an improved socket factory framework is released by Sun, we will be able to achieve fully transparent integration of robust TCP sockets into existing code. The integration will not require any modifications to the Java core libraries (java.net).

6.4 Performance

The benchmarks used to measure the performance of the robust TCP sockets are taken from IBM's SockPerf socket micro-benchmark suite, version 1.2 [IBM00]. These experiments do not benchmark all aspects of communication with sockets. Nevertheless, they should give an indication of the overhead of the robust TCP sockets with respect to plain TCP sockets. The benchmarks are the following:

TCP_RR A message (request) is sent using TCP to another machine, which

Table 6.1: Java vs. Robust TCP in the three benchmarks.

Benchmark	Robust TCP	Java TCP	Overhead
TCP_RR	8572 tr./s	9061 tr./s	5.7%
TCP_Stream	10785 kB/s	10889 kB/s	0.95%
TCP_CRR	3.34 ms	1.30 ms	157%

echoes it back (response). The TCP connection is set up in advance. The results are reported as a throughput rate of transactions per second, which is the inverse of the request / response round-trip time. The benchmark is run several times with different message lengths. The default length in SockPerf is one byte.

TCP_STREAM A continuous stream of messages is sent to another machine, which continuously receives them. The results are bulk throughputs in kilobytes per second. The TCP_STREAM benchmark is run with several different message lengths. The default message length in SockPerf is 8 kilobytes.

TCP_CRR First, a connection is established between the two machines (connect). Then, a message (request, by default 64 bytes) is sent using TCP, and is replied to (by default 8 kilobytes). This reflects the message size of a typical HTTP query. The costs included in the benchmarks are those of the connection establishment, the data exchange and the closing of the connection.

The benchmarks were run with two PCs running Red Hat Linux 7.2 (kernel 2.4.9). The PCs have Pentium III 766 MHz processors and 128 MB of RAM, and are interconnected by a 100 Base-TX Ethernet. The Java Virtual Machine is Sun's JDK 1.4.0.

The results, as well as the relative performance of the robust TCP sockets versus Java sockets, are summarized in Table 6.1 and Figure 6.6. They show that the overhead of the robust TCP sockets over Java sockets is low (5.7% and 1%) for the TCP_RR and TCP_Stream tests, except for small message lengths in the TCP_Stream test. The overhead for these tests is probably due to (1) the one extra copy of transmitted data into the retransmission buffer at the session layer, and (2) the control message processing.

The TCP_CRR test shows a bigger overhead. The overhead is due to the message exchange upon opening the connection (see Figure 6.3). However, this benchmark measures the performance of short-lived TCP connections, whereas robust TCP connections only make sense for long-lived connections – short-lived connections are not likely to break. For this reason, we did not put any effort into optimizing for the TCP_CRR benchmark. A pos-

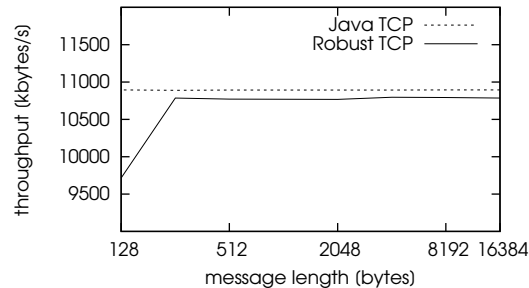


Figure 6.6: Results of the TCP Stream benchmark.

sible optimization is to wait for the first data packet such that the session layer messages can be piggybacked.

6.5 Related work

We start with papers about fault-tolerant TCP connections. Zhang and Dao describe *persistent connections* [ZD95], which can recover from broken transport layer connections, just like our robust connections. As ours, their prototype is also implemented in a library on top of sockets. However, Zhang and Dao both (1) have a more ambitious goal, and (2) do not meet our requirements. Zhang and Dao provide connections where the transport layer endpoints might change their location and/or identity. For example, one endpoint might be a mobile device that migrates, or a process that crashes and then recovers. As the goal is more ambitious, the solution is more complex: it involves an addressing scheme distinct from TCP addressing and a name service used to store information about endpoints. On the other hand, data loss is possible if a connection breaks in an unanticipated manner, while our protocol avoids this. The authors did not avoid data loss because they focused on connections that break due to process crashes, rather than network problems. In such a setting, a session layer mechanism is not enough to provide exactly-once delivery: some help is needed from the application.

The FT-TCP protocol [ABEK⁺01], and STCP [SXM⁺00] to some extent, also aim at making TCP connections fault-tolerant to the crash of one endpoint (while our protocol makes the connection fault-tolerant to link failures). After the crash, either another node has to take over the connection, or the failed node has to recover. Even though the problem is different from ours, the solutions involves a lot of common tasks: buffering data and synchronizing the new node to the state of the stream with the help of the buffered data. A difference is that FT-TCP and STCP require changes in the kernel, as they augment or modify the transport layer. An interesting point

in FT-TCP is that the other (non-fault tolerant) endpoint of the connection runs TCP without any changes: we cannot provide this property, though, as a broken connection affects both endpoints.

The protocols in [BS97, RM98] adapt TCP to wireless environments. Connectivity can be lost in such environments for a long time. The solutions usually passivate the TCP connection when connectivity is lost, to avoid that TCP reacts to this condition by reducing the size of its congestion window or by breaking the connection. These protocols necessitate changes in the kernel.

Finally, the session layer in the ISO/OSI reference model [ISO96] offers some functionality to re-establish broken transport layer connections. The communicating parties can put *synchronization points* into the session layer stream, and it is possible to recover the state of the stream at these synchronization points later. It is the application's responsibility to set synchronization points and to buffer data that might need to be retransmitted. Our solution accomplishes exactly these tasks, making synchronization and buffering transparent to the application.

6.6 Discussion

This chapter presented robust TCP connections for fault tolerant distributed computing. Robust TCP connections, unlike TCP connections, address link and partition failures in a manner adequate for a range of applications. Robust TCP connections never break if connectivity is lost.

We implemented robust TCP connections as a session layer protocol on top of TCP that ensures reconnection, and provides exactly-once delivery for all transmitted data. Our Java prototype has less than 10% overhead on TCP sockets with respect to the most important performance features. It can be easily integrated into existing applications.

Additionally, the interface of the robust TCP connections can be extended in order to provide information about the state of session connections to the application. Useful information includes the number of bytes sent but not acknowledged, the time elapsed since the last send operation whose data was not acknowledged, and the duration for which a receive (or send) operation has been blocked. Yet another idea is to add an operation that allows the application to passivate the connection when there is no need to send data over a long period. A passivated connection uses fewer resources: in particular, the associated TCP connection would be closed.

Comparing atomic broadcast algorithms in a local area network

In Chapter 4, we presented a new token based atomic broadcast algorithm using failure detectors. We compared this algorithm to Chandra-Toueg's atomic broadcast reduction to consensus (using two different failure detector based consensus algorithms) in a simulated environment. The simulation model that was considered in Chapter 4 focuses on the contention on two main system resources: the network and the processors. The model does however not take into account other important factors that affect the performance of the algorithms, such as the size of the messages or the time needed to serialize (or marshal) these messages in order to send them on the network.

For this reason, the experimental evaluation of the algorithms in a real environment is important when assessing their performance. This chapter starts by comparing the performance of the token based atomic broadcast algorithm with the Chandra-Toueg algorithm (coupled with the Chandra-Toueg or Mostéfaoui-Raynal consensus algorithms) in a local area network. We focus on the case of a system without any process failures and examine the situations where (1) no suspicions occur and those where (2) wrong suspicions occur repeatedly. Situation (2) assesses one of the desired properties of the new token based algorithm: a good performance in a system with frequent wrong failure suspicions.

The second part of the experimental evaluation compares our new token based algorithm using failure detectors to another token based atomic broadcast algorithm using group membership to tolerate failures. Token based atomic broadcast algorithms published in the past are known for their high achievable throughput. This second part thus focuses on assessing the second desired property of our new failure detector based algo-

rithm: its behavior when high throughputs are reached.

The chapter is structured as follows: Section 7.1 presents the algorithms that are considered in this performance study and briefly recalls their expected (analytical) performance. The metrics and setup issues of the performance evaluation are then presented in Section 7.2. The results of the performance study are then presented in Section 7.3 in two parts: Section 7.3.1 compares the performance of the Chandra-Toueg atomic broadcast algorithm with our new token based algorithm and in Section 7.3.2, the comparison is done with a moving sequencer (token based) algorithm using group membership.

7.1 Algorithms

We now present the two atomic broadcast algorithms that are compared with the token based atomic broadcast algorithm presented in Chapter 4. The first algorithm uses failure detectors (indirectly), whereas the second one uses group membership for fault tolerance. As in [USS03], all the algorithms are optimized (1) for runs without failures and without suspicions, (2) to minimize the latency when the load on the system is low (rather than minimizing the number of sent messages) and (3) to tolerate high loads.

7.1.1 Chandra-Toueg atomic broadcast algorithm

The atomic broadcast algorithm proposed by Chandra and Toueg [CT96] reduces atomic broadcast to a sequence of consensus. The algorithm is shortly reminded below (the details of the algorithm are presented in Appendix A.3.1).

Whenever a message m is *abroadcast*, it is first reliably broadcast to all processes. The order of the *abroadcast* messages that have not yet been *adelivered* is then determined by consecutive consensus executions $1, 2, \dots$. Each consensus execution is performed on a set of messages (the proposal of process p_i is the set of messages that have been *abroadcast* but that p_i has not *adelivered* yet, the decision of consensus is one of these sets). To *adeliver* a message m that is *abroadcast*, the algorithm thus needs one reliable broadcast and one consensus execution. The cost (in terms of communication steps and sent messages) of *adelivering* an application message thus depends on the choice of the underlying consensus and reliable broadcast algorithms.

In our performance study, we consider the reliable broadcast algorithm presented in [CT96], that requires one communication step and n^2 messages per reliable broadcast. Furthermore, we consider the two same consensus implementations that were already used in the simulated performance study in Section 4.1.6. Both algorithms use an unreliable failure

detector $\diamond S$ [CT96] to solve consensus and require at least a majority of correct processes to reach a decision. The characteristics of the two algorithms are shortly recalled in the following paragraphs.

7.1.1.A Chandra-Toueg consensus

The Chandra-Toueg algorithm solves consensus using a centralized communication scheme. A coordinator collects the estimates of all processes and proposes a value. All processes then acknowledge this proposal to the coordinator or refuse it if the coordinator is suspected. If the proposal is accepted, the coordinator reliably broadcasts the decision to all processes.

If neither failures nor suspicions occur, this algorithm requires $2n$ messages and one reliable broadcast to reach a decision. The decision is received after 3 communication steps by all processes (2 in the case of the coordinator). The details of the algorithm are presented in Appendix A.2.1.

7.1.1.B Mostéfaoui-Raynal consensus

The Mostéfaoui-Raynal algorithm solves consensus using a decentralized communication scheme. Again, a coordinator collects the estimates of all processes and proposes a value. This time, all processes retransmit this proposal to all other processes or send an invalid value (\perp) if the coordinator is suspected. Any process that receives a majority of acknowledgments decides and informs the other processes of its decision.

If neither failures nor suspicions occur, this algorithm requires $2n^2$ messages to reach a decision. The decision is received after 2 communication steps by all processes (or a single step in the case of non-coordinator processes if $n = 3$). The details of the algorithm are presented in Appendix A.2.2.

7.1.2 Moving sequencer atomic broadcast algorithm

The second uniform atomic broadcast algorithm that is considered in this performance study orders the *abroadcast* messages by using a moving sequencer (*i.e.* a token based mechanism) and is based on the algorithms described in [CM84, CM95, CMA97, DSU04, WMK94]. The algorithm does not tolerate failures directly and requires an underlying group membership service to exclude faulty processes. Its communication pattern is however similar to that of the *TokenFD* algorithm. We assume that at least a majority of the processes in the current view do not crash before installing the next view. The following paragraphs briefly describes the algorithm, whereas a detailed presentation is in Appendix A.3.2.

When a message m is *abroadcast*, it is first sent to all processes, including the sequencer (the process that currently holds the token). The sequencer

then assigns an order to all messages it receives and informs all the other processes of the sequence numbers assigned to the new messages. Furthermore, this also passes the token to the next process which then acts as the sequencer for the next batch of unordered messages.

Since we consider *uniform* atomic broadcast (and in particular its uniform total order property), a process cannot *adeliver* a message m as soon as it knows its sequence number. Indeed, imagine that a faulty process p is the sequencer and assigns the next sequence number to a message m . If p immediately *adelivers* m and crashes shortly after, the other processes might never know that a sequence number was assigned to m . This in turn implies that when p is later excluded from the group (by the group membership service), the other processes can *adeliver* another message instead of m and thus violate the uniform total order property of atomic broadcast.

As a consequence, the token also transports the set of assigned sequence numbers that each process knows of. Whenever a sequence number s is acknowledged by a majority of processes, the messages that were ordered in batch number s can be *adelivered*. This ensures that the sequence numbers of all *adelivered* messages are not lost in case of process crashes. Finally, *adelivered* messages are garbage collected only once they have been acknowledged by all processes.

For an *abroadcast* of a message m , the moving sequencer atomic broadcast algorithm sends $n \cdot (1 + \lceil \frac{n+1}{2} \rceil)$ messages and requires $1 + \lceil \frac{n+1}{2} \rceil$ communication steps for all processes to decide.¹

7.2 Elements of our performance study

The following paragraphs describe the benchmarks (*i.e.* the performance metrics, the workloads and the faultloads) that were used to evaluate the performance of the four implementations of atomic broadcast (three atomic broadcast algorithms, one of which uses two different consensus algorithms). Similar benchmarks have been presented in [Urb03, USS03]. The four algorithms that are compared are noted *TokenFD* (token based algorithm using failure detectors, presented in Chapter 4), *CT* (Chandra-Toueg's atomic broadcast with Chandra-Toueg's $\diamond S$ consensus), *MR* (Chandra-Toueg's atomic broadcast with Mostéfaoui-Raynal $\diamond S$ consensus) and *MovingSeq* (token based atomic broadcast algorithm using group membership to tolerate failures).

¹By refraining from sending the token to all processes after each token possession, the algorithm can be adapted so that only $3n + \lceil \frac{n+1}{2} \rceil - 3$ messages are needed for all processes to *adeliver* an *abroadcast* message. However, in this case, the last process to *adeliver* only does so after $n + \lceil \frac{n+1}{2} \rceil - 1$ communication steps. Since all algorithms minimize the latency when the throughput is low, the approach necessitating more messages was chosen.

7.2.1 Performance metrics and workloads

The performance metric that was used to evaluate the algorithms is the latency of atomic broadcast. For a single atomic broadcast, the latency L is defined as follows. Let t_a be the time at which the *abroadcast*(m) event occurred and let t_i be the time at which *adeliver*(m) occurred on process p_i , with $i \in 0, \dots, n - 1$. The latency L is then defined as $L \stackrel{\text{def}}{=} (\frac{1}{n} \sum_{i=0}^{n-1} t_i) - t_a$. In our performance evaluation, the mean for L is computed over many messages and for several executions. 95% confidence intervals are shown for all the results.

Other metrics could have been used, such as the *early* latency, where the smallest measured latency among all processes is used (as opposed to the mean in our case), and which was used in [USS03, UHSK04]. However, such a metric has a strong bias towards the lowest *adelivery* times, which is rather favorable to token based atomic broadcast algorithms.

The latency L is measured for a certain workload, which specifies how the *abroadcast* events are generated. We chose a simple symmetric workload where all processes send atomic broadcast messages² at the same constant rate and the *abroadcast* events come from a Poisson stochastic process. The global rate of atomic broadcasts is called the *throughput* T , which is expressed in messages per second (or *msgs/s*).

Furthermore, we only consider the system in a stationary state, when the rate of *abroadcast* messages is equal to the rate of *adelivered* messages. This state can only be reached if the throughput is below some maximum threshold T_{max} . Beyond T_{max} , some processes are left behind. We ensure that the system stays in a stationary state by verifying that the latencies of all processes stabilize over time.

7.2.2 Faultloads

The faultload specifies the events related to process failures that occur during the performance evaluation [KMA02, Urb03]. In our experiments, the faultload focuses on the process crashes and the behavior of the unreliable failure detectors. We evaluate the atomic broadcast algorithms in the *normal-steady* and *suspicion-steady* faultloads [Urb03] which are presented below.

Normal-steady In the *normal-steady* faultload, only runs without process failures or wrong suspicions are considered. The parameters that influence the latency are n (the number of processes), the algorithm (*TokenFD*, *CT*, *MR* or *MovingSeq*), and the throughput.

²The atomic broadcast messages do not contain any payload, in order to reach the maximum possible performance when comparing the algorithms

Suspicion-steady In the *suspicion-steady* faultload, no processes fail, but wrong suspicions occur. These wrong suspicions can occur in two ways: (1) by using a real failure detector implementation and, for example, setting very low timeout values or (2) by simulating the behavior of the failure detector modules.

The first approach, although closest to the implementation of a system in practice, has several drawbacks that make it unsuitable for our performance evaluation: first of all, the wrong suspicions are difficult to reproduce. Indeed, they depend on the network utilization, packet loss and the processing taking place, among others. Thus, it is difficult, if not impossible, to evaluate the different algorithms in identical (or at least similar) conditions. The second drawback is that the rate of wrong suspicions is difficult to predict. Thus, setting up experiments that evaluate the performance of the algorithms for different wrong suspicion rates is hardly feasible.

The second approach, which simulates the failure detector modules, does not have any of the drawbacks presented above. Furthermore, the quality of service of the simulated failure detectors is easily reproducible and customized. In [CTA02], Chen, Toueg and Aguilera present a model of the quality of service of failure detectors which we use here for the simulated wrong suspicions in the *suspicion-steady* faultload.

The two quality of service metrics presented in [CTA02] that apply to the (failure free) *suspicion-steady* faultload are presented in Figure 7.1 and detailed below:

- The *mistake recurrence time* T_{MR} : The time between two consecutive mistakes (the failure detector module on process q wrongly suspects process p).
- The *mistake duration* T_M : The time needed to correct the mistake of the failure detector (the time needed for q to trust p again).

These two quality of service metrics are random variables that are associated with each pair of processes (where one process monitors the other). In the *CT* and *MR* algorithms, each process monitors the $n - 1$ other processes and we thus have $n \cdot (n - 1)$ failure detectors in the sense of [CTA02]. In *TokenFD*, each process only monitors its predecessor, and we thus have n failure detectors in this second case.

As in [Urb03], and to keep the model as simple as possible, we consider that the two random variables associated with each failure detector are all independent and identically distributed. Furthermore, the random variables T_{MR} and T_M both follow an exponential distribution with a (different) constant parameter. This model is a starting point and does not take into account the correlation that exists between the failure detectors in a real system (*i.e.*, in *CT* and *MR*, if q suspects p , other processes probably

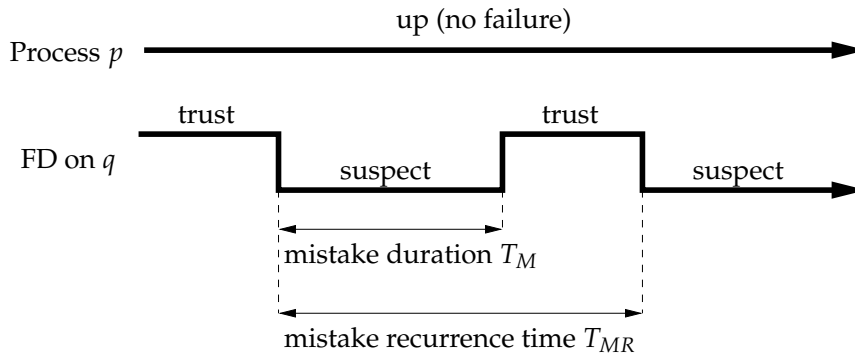


Figure 7.1: Quality of service model of a failure detector in the *suspicion-steady* faultload. Process q monitors process p .

also suspect p). This correlation is less pronounced in *TokenFD* than in both other algorithms, since each process is only monitored by a single other process.

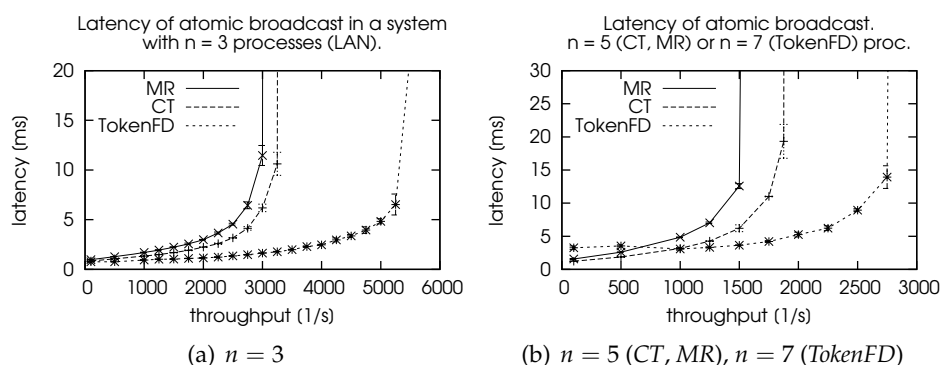
Finally, this simulated failure detector model does not put any load on the system (since no messages are exchanged between the failure detector modules). However, since in a real failure detector implementation, a good quality of service can often be achieved without sending messages frequently, this trade-off is acceptable.

7.2.3 Implementation framework and issues

The algorithms are implemented in Java, using the Neko framework. The implementation of the *CT*, *MR* and *TokenFD* algorithms for the experimental evaluation is the same as the one used for the simulated evaluation in Section 4.1.6. Furthermore, in the experimental setup, all processes are connected pair-wise through TCP channels³

Finally, we enabled Java's *Just-In-Time (JIT)* compilation [CFM⁺97], that transforms often-executed Java bytecode into native machine code. The JIT compilation process adversely affects the performance measurements, since the execution of the code is interrupted while it is compiled. To avoid any interference between the measurements and the JIT compilation, each execution was preceded by a *warm-up* phase that lasted long enough for the Java Virtual Machine to compile the parts of the code that would be executed often during the performance measurements.

³We also tested the algorithms with a simple UDP and IP multicast implementation of reliable channels. The results were however comparable or worse than with the TCP implementation, due to serialization issues.

Figure 7.2: Latency vs. throughput with a *normal-steady* faultload

7.2.4 Evaluation environment

The experiments were executed on a local area network of nodes with a Pentium 4 processor, model number 630, at 3 GHz and with 2 MB of L2 cache. The size of the main memory on each node is 1 GB and the nodes are interconnected by a single Gigabit Ethernet switch. The round-trip time between two nodes is approximately 0.1 ms. All nodes run a SuSE Linux distribution (with a 2.6.11 kernel) and Sun's Java 1.5.0_05 64-bit Server Virtual Machine.

7.3 Results

7.3.1 Comparing failure detector based implementations

We start by comparing the performance of the three failure detector based atomic broadcast implementations (*TokenFD*, *CT* and *MR*) with two faultloads: *normal-steady*, where good runs without failures nor suspicions are considered (Section 7.3.1.A) and *suspicion-steady*, where processes do not crash but wrong suspicions occur (Section 7.3.1.B).

7.3.1.A Normal-steady faultload

The performance of the three algorithms in a system without failures nor suspicions is presented in Figure 7.2. The horizontal axis shows the throughput (in messages per second) that is considered, whereas the latency of the algorithms for a given throughput is shown vertically.

In a system with three processes (Figure 7.2(a)), in which all three algorithms support one failure, *CT* achieves slightly lower latencies than *MR*,

while *TokenFD* reaches the highest throughput and lowest latencies of the three algorithms. There are two main explanations to the difference between *TokenFD* and *CT* or *MR*. First of all, each *abroadcast* message m in *CT* and *MR* results in an *rbroadcast* of m , whereas *TokenFD* only sends m to all processes and thus generates less network traffic and processing costs. Secondly, in *CT* and *MR*, the order of m (i.e., the decision of consensus) is reliably broadcast to all processes, whereas in *TokenFD*, the order is once again simply sent to all processes. The slight difference between *CT* and *MR* is due to the additional messages needed by *MR* to solve consensus.

The situation is similar in a system where two failures are tolerated (Figure 7.2(b)), except that *TokenFD* has a higher latency than *CT* and *MR* when the throughput is low. The explanation is the following: in *TokenFD*, the number of communication steps needed to *adeli*ver a message is equal to $f + 2$ (with f the tolerated failures) and thus, as f increases, the latency of *adeli*ver also increases. In *MR* and *CT* however, the number of communication steps does not depend on f and the latency of the algorithms is less affected by the increase of the system size.

To summarize, in a system with $n = 3$ processes, the latency of atomic broadcast is lower when using *TokenFD* than *CT* or *MR*. Furthermore, *TokenFD* allows a higher rate of *abroadcasts* while maintaining the system in a stationary state. When two failures are tolerated (requiring 5 processes with *CT* and *MR*, 7 processes with *TokenFD*), *CT* and *MR* achieve lower latencies than *TokenFD* when the system load is low. As soon as the load reaches about 1000 *msgs/s*, *TokenFD* again outperforms both other algorithms.

7.3.1.B Suspicion-steady faultload

The performance of the *TokenFD*, *CT* and *MR* algorithms in a system with wrong suspicions (but without process failures) is discussed in the following paragraphs. We start by considering the case where the frequency of these wrong suspicions varies (but the duration of wrong suspicions is fixed) and then consider the dual case, where wrong suspicions occur at a given frequency, but with varying durations.

Impact of the frequency of wrong suspicions Figures 7.3 and 7.4 illustrate the performance of the three algorithms in a system supporting one, respectively two, failures. In each set of four graphs, the two columns group the wrong suspicion durations that are considered (1 and 5 *ms*), whereas each row represents a system load (800 and 1500 *msgs/s*). The horizontal axis of each graph represents the recurrence time of wrong suspicions (the frequency of suspicions is high on the left side and low on the right side of the graph) and the vertical axis once again represents the latency of atomic broadcast.

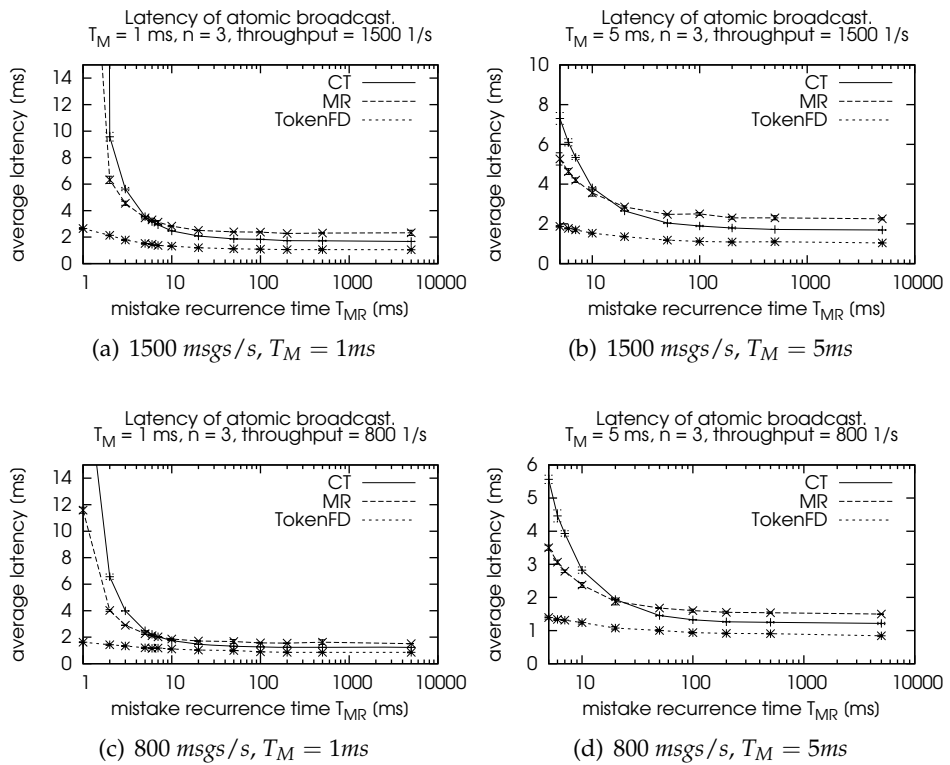


Figure 7.3: Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload in a system with $n = 3$ processes

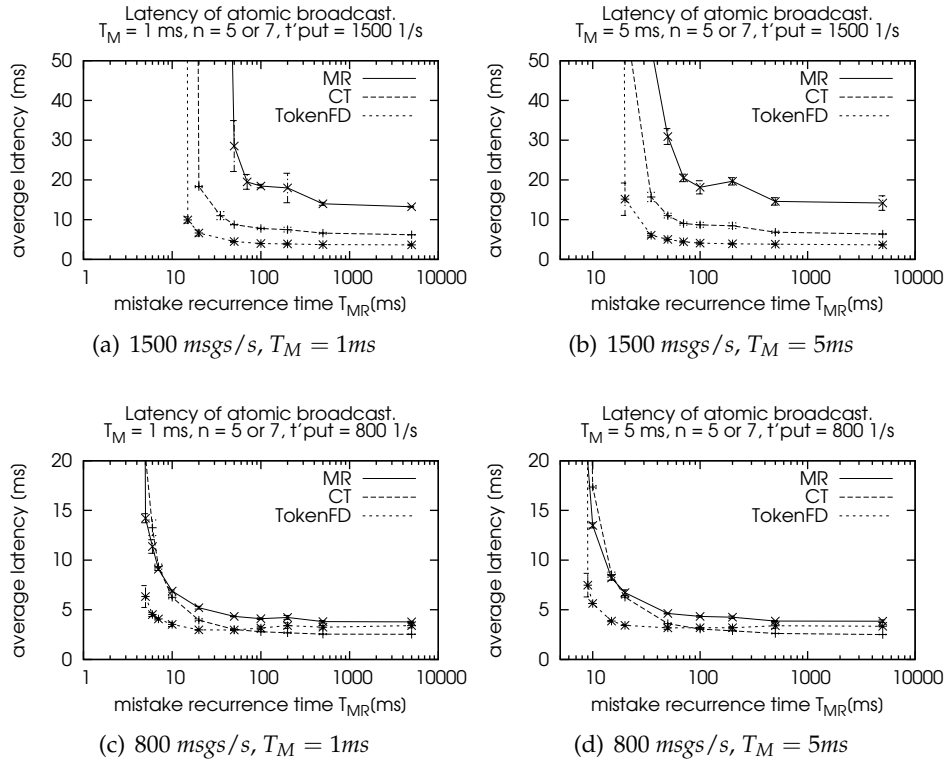


Figure 7.4: Latency vs. mistake recurrence time T_{MR} with a *suspicion-steady* faultload in a system with $n = 5$ (CT, MR) or $n = 7$ (TokenFD) processes

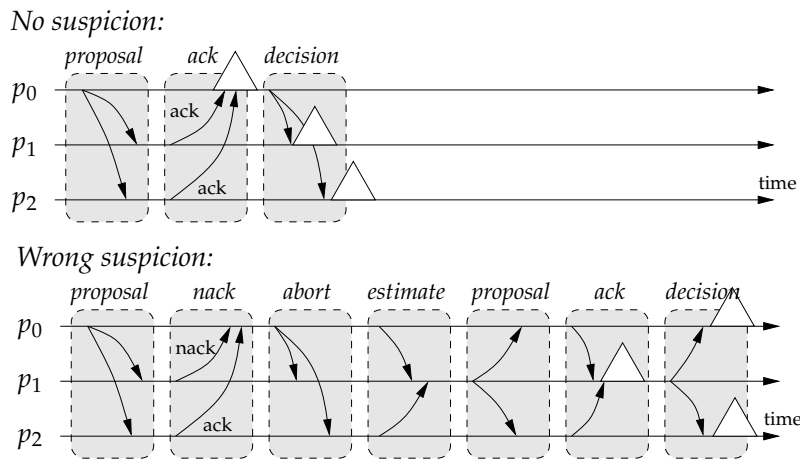


Figure 7.5: Communication pattern of CT in a run without (top) and with (bottom) a wrong suspicion.

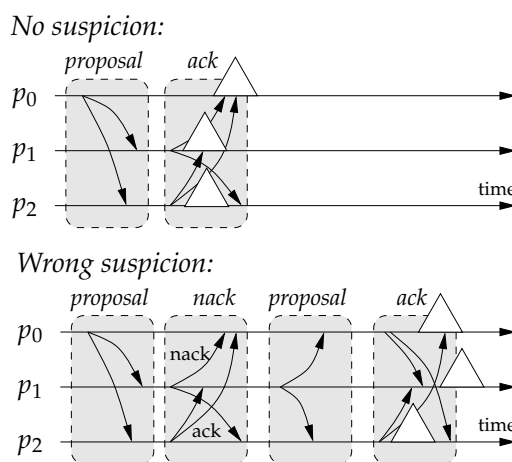


Figure 7.6: Communication pattern of MR in a run without (top) and with (bottom) a wrong suspicion.

In the case of a system with three processes supporting one failure (Figure 7.3), the *TokenFD* algorithm achieves lower latencies than *CT* and *MR*, both in the case of rare wrong suspicions (as T_{MR} , the mistake recurrence time, grows the *suspicion-steady* faultload approaches the *normal-steady* faultload presented previously) and when wrong suspicions occur extremely frequently (in Figures 7.3(a) and 7.3(c), *TokenFD* delivers messages even when wrong suspicions occur on average every millisecond). *TokenFD* achieves lower latencies than *CT* and *MR* in a system with three processes for two reasons: first of all, *TokenFD* can order messages as soon as there exists one process that is not suspected by its successor, whereas in *CT* and *MR*, a consensus execution can be delayed if only a *single* process suspects the coordinator. Secondly, a wrong suspicion is more costly in *CT* and *MR*: if consensus cannot be reached in a given round, the consensus algorithm starts a new round and needs to send at least an additional $4n = 12$ (*CT*, see Figure 7.5) or $n + n^2 = 12$ (*MR*, see Figure 7.6) messages in 4 and 2 additional communication steps respectively⁴. The *abort* communication step in Figure 7.5 is the result of an optimization of the *CT* algorithm that reduces contention in runs without failures nor suspicions (see Appendix A.2.1 for additional details). In the case of *TokenFD*, a wrong suspicion incurs a cost of at least an additional $f + 1 = 2$ messages and one communication step. The cost of a wrong suspicion also explains why

⁴In a system with $n = 3$ processes, the *estimate* step of *CT* shown in Figure 7.5 is skipped since p_1 only needs two estimates (its own and p_0 's) and does not need to wait one communication step for p_2 's estimate. Furthermore, in *MR*, process p_2 can decide at the end of the *nack* step depicted in Figure 7.6 if it receives p_0 's proposal (and its own ack) before p_1 's *nack* value (\perp).

the latency of *MR* is lower than that of *CT* when suspicions are frequent, whereas *CT* outperforms *MR* when (almost) no wrong suspicions occur.

When a system that supports two failures is considered (Figure 7.4), the results are slightly different. Indeed, when the interval between wrong suspicions is low enough — around 20 *ms* (Figures 7.4(a) and 7.4(b)) and 5 *ms* (Figures 7.4(c) and 7.4(d)) when a load of 1500, respectively 800, *msgs/s* is considered — the algorithms cannot *adeliver* messages at the offered load and the latency increases sharply. In the case of *CT* and *MR* in a system with $n = 5$ processes, 4 processes can potentially suspect the coordinator in a round of consensus (up from 2 processes in the previous case of $n = 3$) which increases the chances that wrong suspicions affect consensus. The increased fault tolerance also affects *TokenFD*: indeed, in a system supporting two failures (*i.e.*, $n = 7$), a batch of messages can only be ordered if two consecutive process do not suspect their predecessor.

The ranking between the algorithms is also modified in this second setting: when the load on the system is 1500 *msgs/s*, *CT* achieves lower latencies than *MR*, even as the mistake recurrence time T_{MR} decreases (Figures 7.4(a) and 7.4(b)). The increased system size affects *MR* more than *CT*, due to the $O(n^2)$ messages that *MR*'s consensus uses when the coordinator and the other processes exchange their estimates and acknowledgments (*CT* only needs $O(n)$ messages for this).

Impact of the duration of wrong suspicions Figure 7.7 presents the performance of the three algorithms in a system where wrong suspicions occur on average every 100 *ms*. The duration of these wrong suspicions varies between 1 and 100 *ms* and is shown on the horizontal axis of each graph. The latency of the algorithms is shown on the vertical axis, as previously. In each set of four graphs, each column shows the results for a given system size (supporting 1 or 2 failures). Each row represents a system load (800 and 1500 *msgs/s*).

Once again, when $n = 3$, the latency of *adelivery* of *CT* is lower than that of *MR* when the duration of failures is short (Figures 7.7(a) and 7.7(c), left hand side of the graphs). In such a setting, most of the consensus executions (which only last about one millisecond) terminate without being affected by a wrong suspicion and the performance of the algorithm is thus close to what was observed in the *normal-steady* faultload. As the duration of wrong suspicions increases, a growing number of consensus executions are affected by the suspicions. As discussed above, *CT* pays a higher price than *MR* when a consensus execution is delayed by a suspicion and thus, as the suspicion duration increases, *MR* achieves lower latencies than *CT* (Figures 7.7(a) and 7.7(c), right hand side of the graphs). Finally, when $n = 3$, wrong suspicions do not affect *TokenFD* as much as both other algorithms and *TokenFD* thus achieves the lowest latency of the three algo-

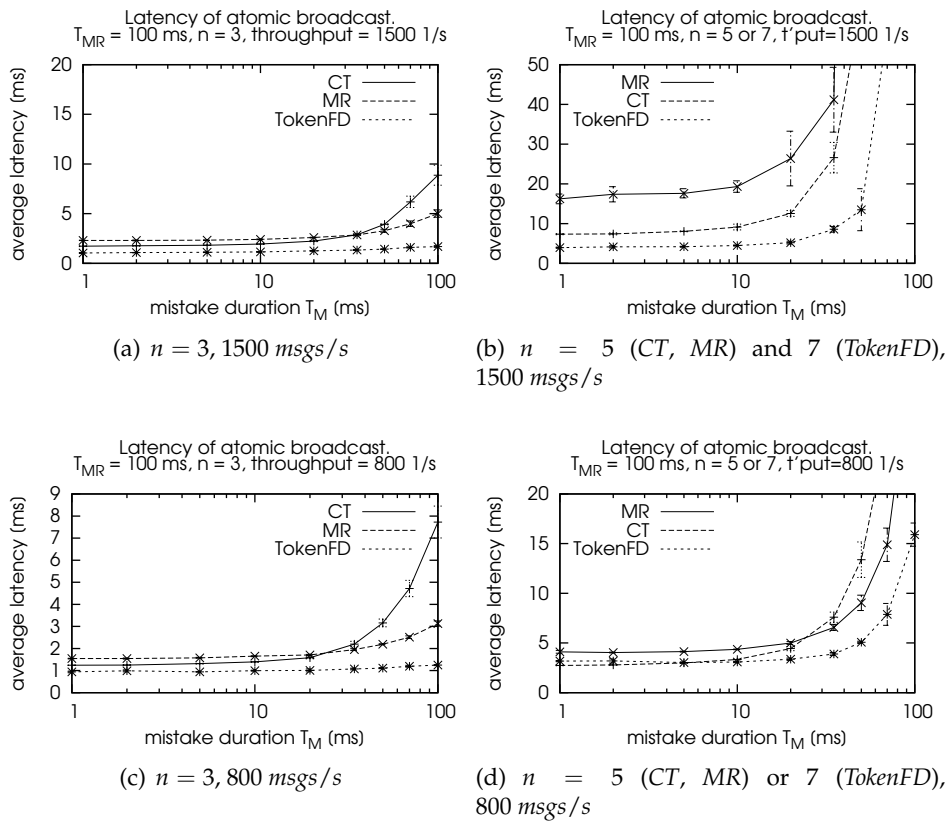


Figure 7.7: Latency vs. mistake duration T_M with a *suspicion-steady* fault-load in a system with a mistake recurrence time of 100ms.

rithms, whether the throughput is moderate (800 *msgs/s*) or higher (1500 *msgs/s*).

In a system with two tolerated failures, presented in the Figures 7.7(b) and 7.7(d), *MR*'s performance drops compared to *CT* and *TokenFD* (again, *MR* sends $O(n^2)$ messages during the proposal and acknowledgment phases of consensus). When the throughput is moderate however (800 *msgs/s*, see Figure 7.7(d)), *MR* achieves lower latencies than *CT* as soon as the duration of the wrong suspicions exceeds 30 *ms*. As in the case $n = 3$, the latency of *TokenFD* is lower than *CT* and *MR* when the duration of wrong suspicions is high, but the performance advantage is less substantial than in a system where a single failure is tolerated.

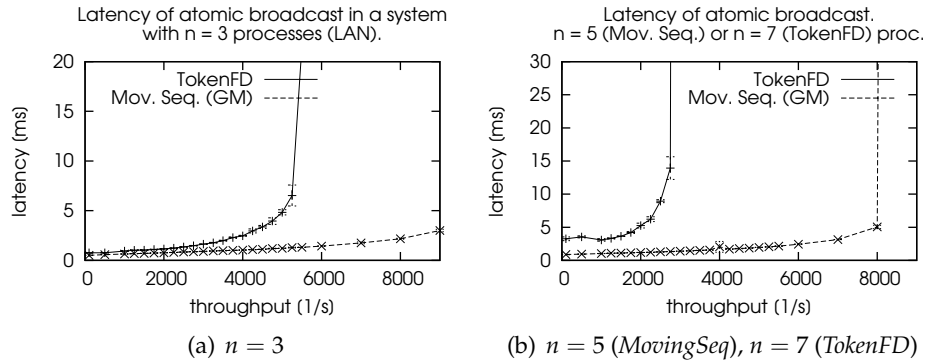
Summary In [USS03], the authors present a performance study of two atomic broadcast algorithms: one based on failure detectors and the other one on group membership. Their performance evaluation confirms that wrong suspicions are better dealt with by failure detectors than by a group membership service that has to carry out several costly operations whenever a suspicion occurs.

The first part of this chapter shows that the performance of the *TokenFD* algorithm is slightly better than both *CT* (the failure detector based algorithm in [USS03]) and *MR* when the interval between wrong suspicions is short. These results confirm that one of the desired properties of the *TokenFD* algorithm holds: wrong failure suspicions do not drastically reduce the performance of the algorithm. This in turn allows an implementation of the failure detector with aggressive timeouts, which consequently allows actual failures to be detected fast. If the failure detector implementation commits a mistake and wrongly suspects a process that is still alive, this mistake does not cost much in terms of performance.

Secondly, the performance evaluation of the three algorithms in the *suspicion-steady* faultload shows a symmetry: the algorithms behave similarly if wrong suspicions occur at an increasing rate but last a constant (on average) amount of time, or if the duration of wrong suspicions increases but that they occur at a constant rate. Indeed, the main factor that influences the performance of the algorithms when wrong suspicions occur is the fraction of time without suspicions. In the case of *CT* and *MR*, this then corresponds to the fraction of consensus executions that are unaffected by wrong suspicions.

7.3.2 Comparing token based implementations

The second part of this chapter compares two token based atomic broadcast: *TokenFD* (which uses failure detectors) and *MovingSeq* (which uses group membership). Token based algorithms are known for the high through-


 Figure 7.8: Latency vs. throughput with a *normal-steady* faultload

put that they achieve. The second part of this chapter thus focuses on the performance of the algorithm in the *normal-steady* faultload, and assesses the second desired property of the *TokenFD* algorithm: high throughput.

Figure 7.8 presents the latency of the *TokenFD* and *MovingSeq* atomic broadcast algorithms as a function of the throughput in a system supporting a single failure (Figure 7.8(a)) and two failures (Figure 7.8(b)). The results in both systems are presented and analyzed in the following paragraphs.

7.3.2.A One tolerated failure

In a system with three processes, presented in Figure 7.8(a), both algorithms achieve similar results when the throughput is moderate (until about 3000 *msgs/s*). For higher loads however, the performance of *TokenFD* degrades faster than *MovingSeq* and above 6000 *msgs/s*, the latency of *TokenFD* does not stabilize (the rate of message *adelivery* is lower than the rate of *abroadcasts*). The *MovingSeq* algorithm, on the other hand, still achieves stable latencies at 9000 *msgs/s*. This behavior is explained by several factors that are detailed in the following paragraphs.

After a single communication step (*i.e.*, one token transmission) of *MovingSeq*, all the processes that receive the token can *adeliver* a message, as shown in Figure 7.9(b) (at the end of the first communication step, processes p_1 and p_2 both *adeliver* message batch 1). Indeed, to *adeliver* a message m , a process must ensure that at least $f + 1$ processes agree on the order of m . When $n = 3$ (and $f = 1$), the $f + 1$ processes are the token sender and the token receiver. In the case of *TokenFD* (Figure 7.9(a)), only the immediate successor of the token sender can *adeliver* the message (since the votes for the proposal in the token are only incremented if there is no gap in the token circulation). The other $n - 1$ processes then get this infor-

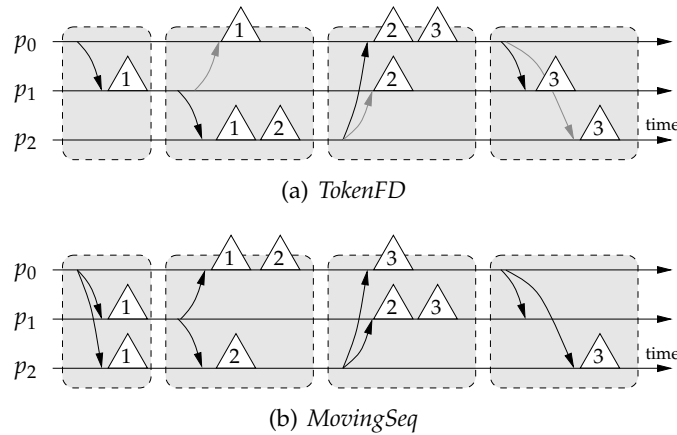


Figure 7.9: Communication patterns of *TokenFD* and *MovingSeq* in good runs. Each triangle with a number i denotes the *adelivery* of the i^{th} message batch.

mation one communication step later. Thus, as Figure 7.9(a) shows, only p_1 *adelivers* message batch 1 after the first communication step, while p_0 and p_2 do so after the second step.

This effect however only explains the small difference between *TokenFD* and *MovingSeq* when the throughput is low. When the throughput increases, other elements affect the performance of the algorithms. First of all, the token transported in *TokenFD* is larger than the one in *MovingSeq*: in *TokenFD*, the token contains the current proposal and a subsequence of the *adelivered* messages. In *MovingSeq*, only the current proposal is transported (the group membership algorithm in *MovingSeq* ensures that all processes have the same *adelivered* sequence if process suspicions or failures occur). The additional data transported by *TokenFD* incurs additional serialization and networking costs. Furthermore, suspicions and failures must be handled by *TokenFD* directly, which creates additional processing costs related to the token circulation (such as comparing the sequence of *adelivered* messages in the token to the local copy of that same sequence). These costs are not present in *MovingSeq*, since the algorithm does not need to handle failures (whenever a failure does occur, it is handled by the group membership service, which ensures that all processes are in a consistent state before installing the next view of the group).

7.3.2.B Two tolerated failures

When two failures are tolerated, in a system with $n = 5$ and $n = 7$ processes for the *MovingSeq* and *TokenFD* algorithms respectively, the results

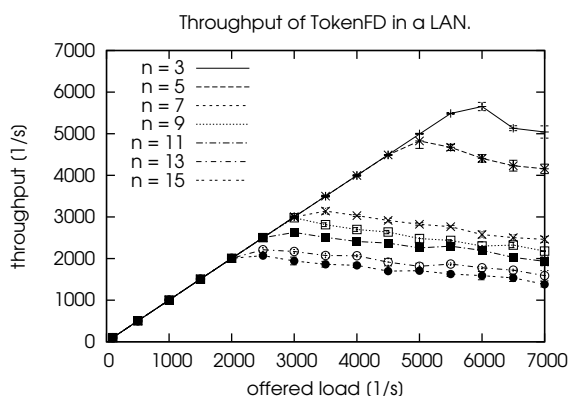


Figure 7.10: Throughput vs. offered load of the *TokenFD* algorithm with a *normal-steady* faultload

are slightly different. Indeed, in this setting, the latency of *TokenFD* is higher than that of *MovingSeq* even for low throughputs, even though both algorithms achieve quasi-constant latencies for throughputs under 1500 *msgs/s*. When the throughput increases, the latency of *TokenFD* stabilizes at increasingly higher values. Finally, above 3500 *msgs/s*, *TokenFD* no longer reaches a stationary state. The same limit for the *MovingSeq* algorithm is above 8000 *msgs/s*.

The difference between both algorithms is again explained by the different data structures that the algorithms operate on. The token in *TokenFD* transports up to two proposals (*i.e.*, f proposals) and a subsequence of the *adelivered* messages, whereas *MovingSeq* again only needs to send the current proposal in the token to all processes. The size of the *MovingSeq* token is thus the same in a system with $n = 3$ or $n = 5$ processes, whereas for *TokenFD*, the size of the token increases linearly with f (the token is thus roughly two times larger when $n = 7$ and two failures are tolerated, than when $n = 3$). Finally, since *TokenFD* requires $n = 7$ processes to support two failures (versus $n = 5$ for *MovingSeq*), *TokenFD* needs to inform 6 other processes each time the order of a message batch is decided. *MovingSeq* only needs to inform 4 processes.

7.3.2.C Impact of the number of tolerated failures on *TokenFD*

Figure 7.8(a) shows that the highest throughput achievable by *TokenFD* decreases sharply as the size of the system increases. The following paragraph examines the correlation between the system size and the highest throughput that *TokenFD* can reach. Figure 7.10 presents the throughput of *TokenFD* as a function of the offered system load, for 7 different system sizes

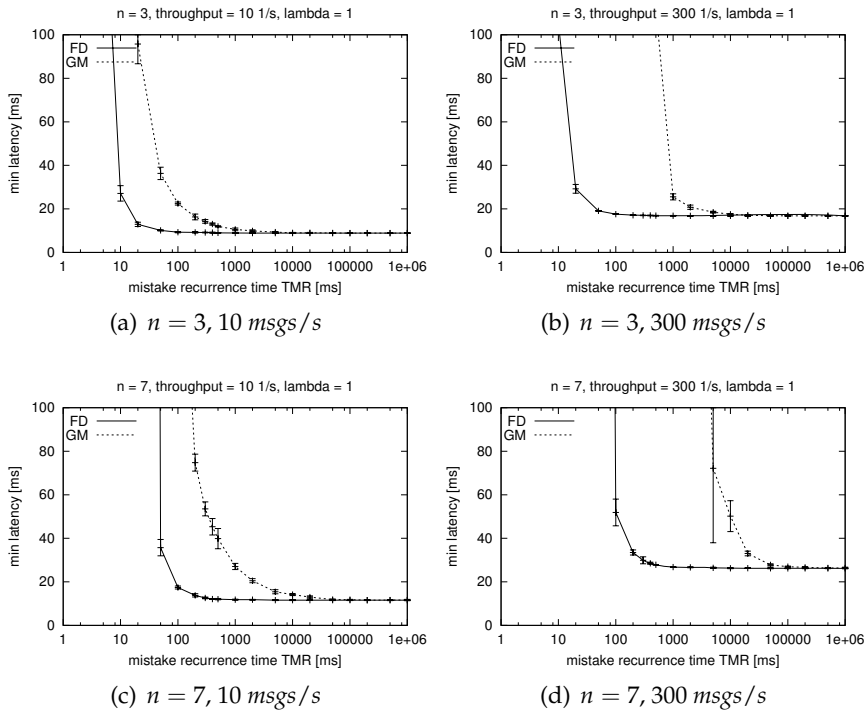


Figure 7.11: Early latency vs. Mistake recurrence time T_{MR} of a group membership and a failure detector based algorithm, simulation results from [USS03]

$n = 3, 5, \dots, 15$. Unlike the previous figures, this figure does not present the latency of atomic broadcast, as it considers system states in which the latency is not necessarily stationary (more precisely, the latency of the system is not stationary as soon as the offered load is higher than the throughput).

The highest throughput of *TokenFD* is heavily influenced by the fault tolerance f of the system, as shown in Figure 7.10. As expected, the smaller systems allow higher throughputs to be reached. What is interesting is the performance gap associated with different levels of the fault tolerance f . Figure 7.10 shows a gap between $n = \{3, 5\}$ (one tolerated failure) and $n = \{7, 9, 11\}$ (two tolerated failures) and $n = \{13, 15\}$ (three tolerated failures). The differences within these three sets are less important. In theory (and in good runs), the latency depends heavily on f (it takes $f + 2$ communication steps to *adeliver* a message), but the maximum throughput should hardly be affected by f , since the token contains several proposals that are being decided upon in parallel. Thus, if the rate of *abroadcasts* is high enough, a batch of messages is ordered at each token possession, independently of the value of f . In practice however, the maximum throughput *is* affected

by the value of f and in particular by the size of the token (which depends on the number of transported proposals, hence f).

7.3.2.D Impact of wrong suspicions on group membership algorithms

In [USS03], the authors compare failure detectors and group membership algorithms. The failure detector algorithm is the *CT* algorithm considered in this chapter and the group membership algorithm uses a fixed sequencer to order messages (and implements its group membership service similarly to the *MovingSeq* algorithm). The evaluation of the algorithms is done by simulation in the Neko framework, with $\lambda = 1$ (sending a message on the network and processing it takes the same time: 1 time unit).

Figure 7.11, taken from [USS03], shows the latency as a function of the mistake recurrence time T_{MR} (and the mistakes are immediately corrected: $T_M = 0$) for two systems ($n = 3$, $n = 7$) and two throughputs (10 *msgs/s* and 300 *msgs/s*). The figure shows that as the frequency of wrong suspicions increases, the latency of the group membership algorithm increases sharply. The *CT* algorithm, on the other hand, is less affected by the wrong suspicions (especially when the throughput is high).

These results, although simulated and not experimental, show the relative cost of a wrong suspicion in failure detector and group membership algorithms. Since the implementation of the group membership service of *MovingSeq* is similar to the one presented in [USS03], it is probable that wrong suspicions affect the performance of *MovingSeq* more than they affect *TokenFD*.

7.4 Discussion

This chapter presented the performance evaluation of four atomic broadcast and consensus implementations in a local area network with a small number of processes. The four algorithms were chosen in order to explore two of the dimensions that characterize atomic broadcast algorithms: (1) the fault tolerance mechanism, which was either failure detectors (*CT*, *MR* and *TokenFD*) or group membership (*MovingSeq*) and (2) the ordering mechanism, which was either token based (*TokenFD*, *MovingSeq*) or a reduction to consensus (*CT*, *MR*). Consequently, we also evaluated if the *TokenFD* algorithm displayed the qualities associated with token based algorithms (high throughput) and failure detector based algorithms (good performance even in the presence of wrong suspicions).

The first part of the evaluation (in systems without failures nor suspicions) showed that the latency of *TokenFD* is lower than that of *CT* and *MR*, especially as the throughput increases. Furthermore, the highest throughput achieved by *CT* and *MR* was 3500 *msgs/s* ($n = 3$) and 2000 *msgs/s*

($n = 5$), whereas *TokenFD* reached 5500 *msgs/s* ($n = 3$) and 3000 *msgs/s* ($n = 7$). These results show that *TokenFD* reaches higher throughputs than other failure detector based algorithms in systems that tolerate one or two failures. Since *TokenFD* requires $f(f + 1) + 1$ processes (*i.e.* $O(f^2)$) to tolerate f failures, its use is mainly targeted at these systems with limited fault tolerance needs.

Secondly, this chapter addressed the case of a system where wrong suspicions occur (but without process failures). Again, *TokenFD* achieved lower latencies than *CT* and *MR* when the rate of wrong suspicions increased, in a system supporting one or two failures. This confirms one of the expected qualities of *TokenFD* : handling frequent wrong suspicions while limiting the impact on performance. In [USS03], the authors compared *CT* with a group membership algorithm (using a fixed sequencer to order messages) and showed that, in simulation, *CT* outperformed the group membership algorithm when the recurrence time between wrong suspicions was low. In this chapter, we showed (in a real experimental setting, not in simulation) that *TokenFD* achieves yet lower latencies than *CT* when wrong suspicions are frequent.

Finally, we compared *TokenFD* to another token based algorithm, *MovingSeq*, that relies on a group membership service to support process failures. This comparison showed that *TokenFD* does not reach as high throughputs as *MovingSeq* in runs without process failures or suspicions. Since both algorithms have similar communication patterns, the performance gap is explained in part by the additional processing costs and the larger messages needed to support failures directly in *TokenFD*. These costs are reduced in *MovingSeq*, since the algorithm delegates the handling of failures to an underlying group membership service. However, when wrong suspicions occur, a costly protocol is needed in *MovingSeq* to ensure that all processes have the same view of the group membership.

As mentioned above, *TokenFD* was designed with two properties in mind: good performance when wrong suspicions are frequent (with the use of unreliable failure detectors to support process crashes) and high throughput when no failures or suspicions occur (with the use of a token to reduce network contention). This chapter confirms that the first property is clearly implemented by *TokenFD*. In the case of the high throughput property, the experimental evaluation of *TokenFD* shows that it sustains higher system loads than *CT* and *MR*, the two other failure detector based algorithms, but falls behind *MovingSeq*, the group membership and token based algorithm. The *MovingSeq* algorithm however has other shortcomings, especially when wrong suspicions of correct processes occur.

Modeling and validating the performance of atomic broadcast algorithms in high latency networks

Chandra and Toueg introduced the concept of failure detectors in [CT96]. Since then, several atomic broadcast [ESU04a] and consensus [CT96, MR99, Lam98] algorithms based on failure detectors have been published.

The performance of these algorithms is affected by a trade-off between the number of communication steps and the number of messages needed to reach a decision. Some algorithms reach decisions in few communication steps but require more messages to do so. Others save messages at the expense of additional communication steps (to diffuse the decision to all processes in the system for example). This trade-off is heavily influenced by the message transmission and processing times. When deploying an atomic broadcast algorithm, the user must take these factors into account in order to choose the algorithm that is best adapted for the given network environment.

The performance of these algorithms has been evaluated in several environments, both real as in [ESU04a, CUBS02] and in Chapter 7, and simulated [UHSK04, USS03]. However, these evaluations are limited to a symmetrical setup: all processes are on the same local area network and have identical peer-to-peer round-trip times. Furthermore, they only consider low round-trip times between processes (and thus comparatively high message processing costs): a setting which is favorable to algorithms that limit the number of sent messages, at the expense of additional communication steps.

Contributions In this chapter, we model and evaluate the performance of three atomic broadcast algorithms using failure detectors with three different communication patterns (the first based on a reduction to a centralized consensus algorithm [CT96], the second based on a reduction to a decentralized consensus algorithm [MR99] and the third one, the new token based algorithm presented in Chapter 4) in wide area networks. We specifically focus on the case of a system with three processes — *i.e.*, supporting one failure — where either (i) all three processes are on different locations and (ii) the three processes are on two locations only (and thus one of the locations hosts two processes). The algorithms are evaluated with a large variation in link latency (*e.g.*, round-trip times ranging from 4 to 300 ms).

We propose a simple model of the wide area network to analytically predict the performance of the three algorithms. The experimental evaluation confirms that the model correctly predicts the performance for average system loads and for all round-trip times that we considered.

The experimental evaluation of the algorithms leads to the following conclusions. First, the number of communication steps of the algorithms is the predominant factor in wide area networks, whether the round-trip time is high (300 ms) or, more surprisingly (since message processing times are no longer negligible), if it is low (4 ms). The performance ranking of the three algorithms is the same in all the wide area networks considered, despite the two orders of magnitude difference between the smallest and largest round trip times. Secondly, the performance of each of the algorithms heavily depends on setup issues that are orthogonal to the algorithm (typically the choice of the process that starts each iteration of the algorithm, which can be always the same process, or which can shift from one process to another at each iteration). These setup issues also determine the maximum achievable throughput. Finally, when comparing the measurements presented in Chapter 7 with those in this chapter, the performance ranking of the three algorithms is fundamentally different in a wide area network than in a local area network, as expected.

The chapter is structured as follows. Section 8.1 discusses the motivation for evaluating the atomic broadcast algorithms in wide area networks. Sections 8.2 and 8.3 present respectively the system model and the performance metrics that are used. The evaluation of the algorithms is then presented in Section 8.4 (analytical evaluation) and Section 8.5 (experimental evaluation). Finally, Section 8.6 concludes the chapter.

8.1 Motivation and Related Work

In [FLP85], the authors show that consensus cannot be solved in an asynchronous system with a single crash failure. Several extensions to the asynchronous model, such as failure detectors [CT96], have circumvented this

impossibility and agreement algorithms [CT96, MR99, ESU04a] have been developed in this extended model.

The performance of these atomic broadcast algorithms is evaluated in different ways. Usually, the formal presentation of the agreement algorithms is accompanied by analytical bounds on the number of messages and communication steps that are needed to solve the problem [CT96, MR99, VR02]. This coarse-grained evaluation of the performance of the algorithms is however not sufficiently representative of the situation in a real environment.

To get a more accurate estimation of the performance of the atomic broadcast algorithms, they have often been evaluated in local area networks [ESU04a, CUBS02], simulated in a symmetrical environment where all links between processes have identical round-trip times [UHSK04, USS03] or evaluated in hybrid models that introduce artificial delays to simulate wide area networks [VR02].

Although these performance evaluations do provide a representative estimate of the performance of atomic broadcast on a local area network, they cannot be used to extrapolate the performance of the algorithms on a wide area network, where the ratio between communication and processing costs is completely different. Furthermore, evaluating the performance of atomic broadcast on wide area networks is not only of theoretical interest. As [LKPMJP05] shows, it is feasible to use atomic broadcast as a service to provide consistent data replication on wide area networks. In this chapter, we model the performance of these algorithms *and* validate this analysis by experimentally evaluating the algorithms in wide area networks.

We now discuss the central trade-off that explains the impact of network latency on the performance of atomic broadcast algorithms.

8.1.1 The trade-off between number of messages and communication steps

The processes executing the atomic broadcast algorithms that we consider in this chapter communicate with each other to agree on a common message delivery sequence. To do so, they need to exchange a minimum number of messages in a number of communication steps. There is here a trade-off on the number of communication steps and the number of sent messages. Usually, a higher number of messages enables the algorithm to reach a decision in fewer communication steps and vice-versa.

Each communication step has a cost. Indeed, each additional communication step induces a delay on the solution to the problem. This cost is typically low in a local area network, whereas it increases with the latency in a wide area network.

Sending messages also has a cost. Whenever a message is sent, it has

to be handled by the system. This handling includes costs related to algorithmic computations on its content, serialization (*i.e.* transforming the message to and from an array of bytes that is sent on the network) and bandwidth used for the transmission.

These costs characterize the trade-off between the number of messages sent and communication steps needed by the algorithm. If a communication step costs nothing, then the algorithm that sends the least number of messages performs the best. If, on the other hand, a communication step is very expensive, the algorithm that sends most messages (and thus saves on the number of communication steps) has the best performance. In this performance study, several network latencies are studied to evaluate their impact on this trade-off.

8.1.2 Related work

In [ADGS03], the authors study the influence of network loss on the performance of two atomic broadcast algorithms in a wide area network. To do this, the authors combine experimental results obtained on a real network with an emulation of the atomic broadcast algorithms. The scope of the work in [ADGS03] is different from ours: they evaluate the impact that message loss has on the performance of atomic broadcast algorithms whereas we model and evaluate the impact of *network latency* on the relative performance of different algorithms. Furthermore, the experimental performance evaluation in [ADGS03] does not take the processing time of messages into account (the results are based on message logs and emulated algorithms). Arguably, this processing time is negligible when a network with large round-trip times is considered (as was the case in [ADGS03]), but its importance increases as the round-trip times decrease.

Bakr and Keidar evaluate the duration of a communication round on the Internet in [BK02]. Their work focuses on the running time of four distributed algorithms with different message exchange patterns, and in particular, the effect of message loss on these algorithms. Their experiments are run on a large number of hosts (10) and the algorithms that they examine do not allow messages to be lost (*i.e.* an algorithm waits until it has received all messages it is expecting). The scope of [BK02] is similar to ours in that they analyze the relative performance of algorithms with different communication patterns on a wide area network. However, their algorithms are *not* representative of failure detector based atomic broadcast algorithms. Indeed, in the three algorithms we consider, processes never need to wait for messages from *all* the other processes. Thus, if messages from one process are delayed because of a high-latency link, it does not necessarily affect the performance of the atomic broadcast algorithm (whereas it would in [BK02]).

In [VR02], an atomic broadcast algorithm that is specifically targeted towards high latency networks is presented. The authors also evaluate the performance of the algorithm in a local area network with added artificial delays (to simulate the high latency links). The artificial delay is however not sufficient to adequately represent the network links of a wide area network. Indeed, such links are also characterized by a lower bandwidth than local area network links. In our performance measurements, we show that in some cases, the low bandwidth of the wide area links strongly limits the performance of the algorithms that are considered.

Several other papers [USS03, UHSK04, CUBS02, SPMO02, GLPQ06] have studied the performance of atomic broadcast algorithms that use failure detectors or properties related to the spontaneous ordering of messages in local area networks. These papers however, either study the performance of the algorithms in a local area network or through simulation. None of these evaluations adequately models the impact of the high latency links in a wide area network on the performance trade-off between the number of messages that are sent and the number of communication steps needed by the agreement algorithm.

8.2 System model

We consider an asynchronous system of n processes p_0, \dots, p_{n-1} . The processes communicate by message passing over reliable channels and at most f processes may fail by crashing (*i.e.* we do not consider Byzantine faults). A process that never crashes is said to be *correct*, otherwise it is *faulty*. The system is augmented with unreliable failure detectors and is presented in further detail in Chapter 3.

In the following paragraphs, we informally present reliable broadcast, consensus and atomic broadcast (the formal definition of the three problems can be found in Section 3.2). Reliable broadcast and consensus are building blocks for solving atomic broadcast in two of the atomic broadcast algorithms that we consider. In Sections 8.2.2 and 8.2.3, we shortly present the algorithms that are evaluated later.

8.2.1 Reliable broadcast, consensus and atomic broadcast

In the *reliable broadcast* problem, defined by the primitives *rbroadcast* and *rdeliver*, all processes need to agree on a common set of delivered messages. In this chapter, we consider the reliable broadcast algorithm presented in [CT96], which requires $O(n^2)$ messages and a single communication step to *rbroadcast* and *rdeliver* a message m .

Informally, in the *consensus* problem, defined by the two primitives *propose* and *decide*, a group of processes have to agree on a common decision. In this chapter, we consider the two consensus algorithms that use the $\diamond S$ failure detector [CT96] that were considered in the previous chapters (presented in Section 8.2.2).

In the atomic broadcast problem, defined by the two primitives *abroadcast* and *adeliver*, a set of processes have to agree on a common total order delivery of a set of messages. It is a generalization of the reliable broadcast problem with an additional ordering constraint. In this chapter, we consider the two atomic broadcast algorithms (Chandra and Toueg, and *TokenFD*) that are shortly described in Section 8.2.3.

8.2.2 Two consensus algorithms

The first consensus algorithm, proposed by Chandra and Toueg [CT96] and noted *CT*, is a centralized algorithm that requires 3 communication steps, $O(n)$ messages and 1 reliable broadcast for all processes to reach a decision in *good* runs (*i.e.* runs without any crashes or wrong suspicions). The behavior of the *CT* algorithm is detailed in Appendix A.2.1.

The second consensus algorithm, proposed by Mostéfaoui and Raynal [MR99] and noted *MR*, is a decentralized algorithm that requires 2 communication steps and $O(n^2)$ messages¹ for all processes to reach a decision in good runs. The behavior of the *MR* consensus algorithm in good runs and in a system with $n = 3$ processes is detailed in Appendix A.2.2.

On the choice of a coordinator: Both the *CT* and *MR* consensus algorithms use a *coordinator* that proposes the value that is to be decided upon. This coordinator can be any process in the system, as long as it can be deterministically chosen by all processes (based only on information that is locally held by each process). In the analytical and experimental evaluations of these algorithms, we examine how the choice of the first coordinator influences the performance of the algorithms. We also study the case where the first coordinator changes between instance number k of consensus and the next instance $k + 1$.

8.2.3 Two atomic broadcast algorithms

Chandra-Toueg atomic broadcast [CT96]: Figure 8.1(a) shows the communication pattern of Chandra and Toueg’s atomic broadcast algorithm. It requires at least one reliable broadcast and a consensus execution for all processes to *abroadcast* and *adeliver* messages.

¹The *MR* consensus algorithm does not use reliable broadcast as a building block. Instead, reliable diffusion of the decision is ensured by an ad-hoc protocol using n^2 messages.

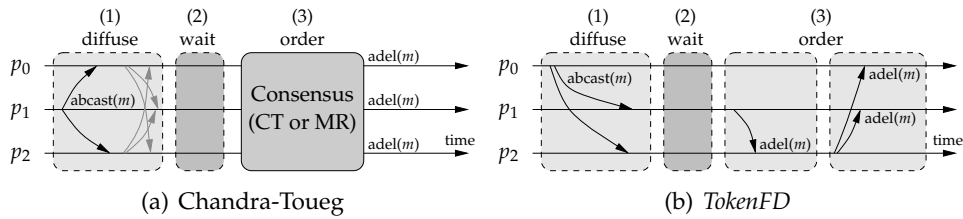


Figure 8.1: Communication pattern of the Chandra-Toueg and *TokenFD* atomic broadcast algorithms in runs without failures or wrong suspicions.

Whenever a message m is *abroadcast*, it is reliably broadcast to all processes (first communication step in good runs). The processes then execute consensus on the messages that haven't been *adelivered* yet (using the *CT* or *MR* algorithm in our case). If m is in the decision of consensus, then m is *adelivered*. The waiting period that is shown in Figure 8.1(a) happens if a consensus execution is already in progress and therefore prevents m from being proposed at once for a new consensus.

Token using an unreliable failure detector [ESU04a]: The token based atomic broadcast algorithm (noted *TokenFD*) presented in Section 4.1 solves atomic broadcast by using an unreliable failure detector noted \mathcal{R} and by passing a token among the processes in the system. It requires three communication steps in a system with $n = 3$ processes and $O(n)$ messages for all processes to *abroadcast* and *adeliver* messages. In runs without failures and suspicions, the *TokenFD* algorithm behaves as in Figure 8.1(b).

Whenever a message m is *abroadcast*, it is sent to all processes (first communication step). Message m is then added to the token that circulates among the processes (in Figure 8.1(b), the token circulates between p_1 and p_2 in communication step 2). After the second communication step, m is *adelivered* by the token-holder which sends an update to all other processes about this delivery (third communication step). Again, the waiting period that is depicted in Figure 8.1(b) only happens if the token is already being sent on the network and therefore prevents m from being ordered immediately.

The two atomic broadcast and two consensus algorithms are representative of a large spectrum of failure detector based algorithms (that require a system size of three processes to support one failure). Indeed, these algorithms all have different communication patterns (centralized, decentralized and token based) and require a varying number of communication steps (between 2 and 4 steps for all processes) and messages (between $O(n)$ and $O(n^2)$) to solve atomic broadcast.

8.3 Performance metrics and workloads

The following paragraphs describe the benchmarks (*i.e.*, the performance metrics and the workloads) that were used to evaluate the performance of the three atomic broadcast algorithms (reduction to *CT* consensus; reduction to *MR* consensus; *TokenFD* algorithm). The benchmarks in [Urb03, USS03, ESU04a] and in the previous chapters are similar to the ones we use here.

8.3.1 Performance metric – latency vs. throughput:

The performance metric that was used to evaluate the algorithms is the latency of atomic broadcast. For a single atomic broadcast, the latency L is defined as follows. Let t_a be the time at which the *abroadcast*(m) event occurred and let t_i be the time at which *adeliver*(m) occurred on process p_i , with $i \in 0, \dots, n - 1$. The latency L is then defined as $L \stackrel{\text{def}}{=} (\frac{1}{n} \sum_{i=0}^{n-1} t_i) - t_a$. In our performance evaluation, the mean for L is computed over many messages and for several executions. 95% confidence intervals are shown for all the results.

8.3.2 Workloads:

The latency L is measured for a certain workload, which specifies how the *abroadcast* events are generated. We chose a simple symmetric workload where all processes send atomic broadcast messages (without any payload) at the same constant rate and the *abroadcast* events follow a Poisson distribution. The global rate of atomic broadcasts is called the *throughput* T . We then evaluate the dependency between the latency L and the throughput T .

Furthermore, we only consider the system in a stationary state, when the rate of *abroadcast* messages is equal to the rate of *adelivered* messages. This state can only be reached if the throughput is below some maximum threshold T_{max} . Beyond T_{max} , some processes are left behind. We ensure that the system stays in a stationary state by verifying that the latencies of all processes stabilize over time.

Finally, we only evaluate the performance of the algorithms in good runs, *i.e.*, without any process failures or wrong suspicions. The latency of the algorithms is measured once the system has reached a stationary state (at a sufficiently long time after the start up). The parameters that influence the latency are n (the number of processes), the algorithm (*TokenFD*, Chandra-Toueg atomic broadcast with *CT* or *MR* consensus) and the throughput.

We specifically focus on the case of a system with three processes, supporting one failure. This system size might seem small. However, atomic broadcast provides strong consistency guarantees (that can be used to implement active replication for example [Sch93a]) and is limited to relatively small degrees of replication. If a large degree of replication is needed, then alternatives that provide weaker consistency should be considered [AM98].

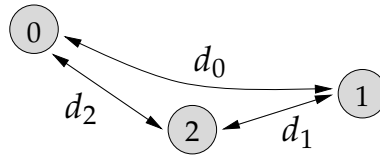
8.4 Modeling the performance of the algorithms

This section discusses the analytical performance evaluation of the two atomic broadcast (and consensus) algorithms in a wide area network. We start by describing the different phases that are common to both atomic broadcast algorithms and then present the two wide area network models that are considered. For the sake of clarity, we only present a partial derivation of the average latency of the Chandra-Toueg algorithm in the simplest wide area network model that we consider. The average latencies of the algorithms in all other settings can be found in Appendices B.1 and B.2. Finally, the predictions of the model are shown alongside the experimental evaluation of the algorithms in Section 8.5.

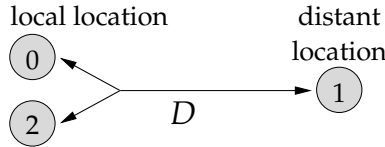
8.4.1 The three phases of atomic broadcast.

Both atomic broadcast algorithms work in three phases, as previously illustrated in Figure 8.1: upon *abroadcasting* a message m , (1) m is sent to all processes and (2) waits to be ordered. Its order is then decided (3), using consensus or directly within the *TokenFD* atomic broadcast algorithm, and m is *adelivered*. The cost of the different phases (and thus the average latency) is of course directly related to the atomic broadcast and consensus algorithms that are used. Furthermore, in all three algorithms, one of the processes, say p_j , has a privileged role: in the *CT* and *MR* algorithms it is the coordinator and in the *TokenFD* algorithm, it is the token holder. The choice of p_j determines the cost of the three phases presented above.

In phase (1), a message m is broadcast by a process p_i to all processes, and in particular to the privileged process p_j , who's in charge of ordering m . The cost of this transmission from p_i to p_j is noted $CostSend_{i,j}$. The cost of the waiting phase (2) depends on the duration of ordering phase (3), since unordered messages cannot be ordered until the end of the currently running consensus instance (*CT* and *MR*) or token circulation (*TokenFD*). The cost of the waiting phase for a message m sent from p_i to the privileged process p_j is noted $CostWait_{i,j}$. Finally, the cost of the ordering phase (3) is



(a) Sending a message between location i and $i + 1$ takes d_i time units.



(b) Sending a message between the local and the distant location takes D time units.

Figure 8.2: Theoretical model of a wide area network with three locations (8.2(a)) or two locations (8.2(b)). The processing times of messages are considered negligible.

only determined by the choice of the privileged process p_j and is noted $CostOrder_j$. The average latency for p_i *abroadcasting* a message m with p_j as privileged process is thus:

$$CostSend_{i,j} + CostWait_{i,j} + CostOrder_j$$

Moreover, all messages sent by a process p_i are not necessarily ordered by the same process p_j . We define $OrderedBy_{i,j}$ to represent the fraction of all messages in the system that are *abroadcast* by p_i and ordered by the privileged process p_j (and we have $\sum_i \sum_j OrderedBy_{i,j} = 1$). Thus, by taking into account all sending processes p_i in the system, we have the average cost of *abroadcasting* a message m :

$$\sum_i \sum_j (CostSend_{i,j} + CostWait_{i,j} + CostOrder_j) OrderedBy_{i,j}$$

8.4.2 Wide-area network with three locations.

Figure 8.2(a) presents the model of a wide area network system with three processes on three different locations. The network latency between location i and location $i + 1$ is noted d_i . Without loss of generality, we assume that $d_0 \geq d_1 \geq d_2$. The model is simplified, in the sense that the processing costs of the messages are considered negligible. This assumption is reasonable if the latencies d_i between locations are much larger than the processing times of the messages on the critical path of the atomic broadcast

algorithms (which is reasonable in a wide area network, but does not hold in a local area network). Furthermore, the model does not take other factors into account, such as the bandwidth of the links or message loss. The analytical expression of the average latency of the three atomic broadcast algorithms in this model can be found in Appendix B.1.

The performance of the algorithms in this model depends heavily on the relationship between the values d_0 , d_1 and d_2 . The analytical comparison between *TokenFD*, *MR* and *CT* with shifting coordinators when d_0 , d_1 and d_2 can take any value is complex and omitted here. In the experimental evaluation of the algorithm, where the values of d_0 , d_1 and d_2 are known, the modeled performance is easily calculated and is presented alongside the experimental results.

8.4.3 Wide-area network with two locations.

The algorithms that are evaluated require a system with at least three processes to tolerate one failure. These three processes can be distributed on up to three different locations. The situation where three locations are used is modeled above and the case where all three processes are on a single location is outside the scope of this chapter, since a wide area network is no longer necessary. The second case, where the processes are on *two* locations, is however interesting: this setup limits the damage due to a catastrophic event at one of the locations and offers the possibility of serving clients from two separate locations (thus reducing the response latency in some circumstances). The model of the two-location system is presented in the following paragraphs.

Figure 8.2(b) presents the model of a system with three processes, one of which is on a distant location. The network latency between the distant location and the local location is noted D . The *two-location model* is a special case of the previous model, with $d_0 = d_1 = D$ and $d_2 = 0$.

We now model the performance of the Chandra-Toueg atomic broadcast algorithm (with *CT* or *MR* consensus) in the case of a fixed initial coordinator p_1 on the distant location, summarized in Table 8.1. The other cases are presented in Appendix B.2.

The cost of the message diffusion phase (1) is the following: the coordinator (on the distant location) receives messages from the two processes on the local location (both with a cost of $CostSend_{0,1} = CostSend_{2,1} = D$) and from itself (with a negligible cost). The average cost for diffusing the message to the distant initial coordinator is thus $\frac{2D}{3}$.

The waiting phase (2) takes the following amount of time: in the *CT* and *MR* consensus algorithms, the coordinator decides after 2 communication steps and atomic broadcast immediately starts a new consensus (if unordered messages are waiting). The cost of these two communication

Table 8.1: Average latency to *adeliver* a message in the two-location wide area network model, using Chandra-Toueg’s atomic broadcast algorithm with *CT* or *MR*’s consensus algorithm and a consensus coordinator on the distant location.

Consensus alg.:	<i>CT</i>	<i>MR</i>
(1) diffusion:	$\frac{2D}{3}$	$\frac{2D}{3}$
(2) waiting:	D	D
(3) ordering:	$\frac{8D}{3}$	$\frac{4D}{3}$
Average latency:	$\frac{13D}{3}$	$3D$

steps is equal to $2D$ time units (*i.e.* one round-trip between the distant and local locations) and the messages thus wait on average $CostWait_{*,1} = D$ time units to be proposed in a consensus (since messages are *abroadcast* following a Poisson process).

Finally, the cost of the *CT* consensus phase (3) is the following. The coordinator can decide after two communication steps and all other processes after three steps. The coordinator p_1 thus decides after $2D$ time units, whereas the processes on both local locations decide one communication step later (thus after a total of $3D$ time units). The average decision duration of phase (3) when the *CT* algorithm is used is therefore $CostOrder_1 = \frac{8D}{3}$.

The cost of the consensus phase (3) using the *MR* algorithm is similar. However, with *MR*, both local locations decide as soon as they receive the coordinator’s proposal and their own acknowledgment, after a total of D time units (*i.e.* they decide *before* the coordinator). As in *CT*, the coordinator decides as soon as it gets an acknowledgment from a local location, after a total time of $2D$. The average latency over all processes is therefore $CostOrder_1 = \frac{4D}{3}$.

After summing up these three phases, the Chandra-Toueg atomic broadcast algorithm using Chandra-Toueg’s consensus algorithm with a fixed initial coordinator on the distant location *adelivers* a message on average $\frac{13D}{3}$ time units after it was *abroadcast*. If the Mostéfaoui-Raynal consensus algorithm is used instead, $3D$ time units are necessary on average to *adeliver* an *abroadcast* message.

8.5 Experimental performance evaluation

In the following section, the experimental performance of the atomic broadcast algorithms presented in Section 8.2 are compared. First, we briefly present the evaluation environments that were considered and then the results that were obtained are presented, analyzed and compared to the

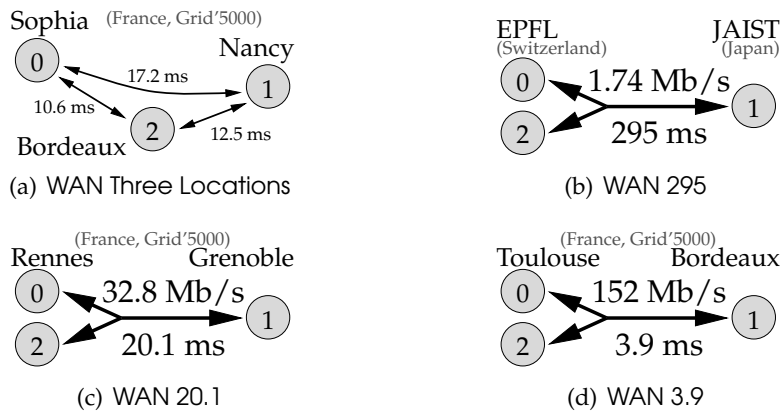


Figure 8.3: Wide area network evaluation environments in decreasing order of round trip times.

analytical evaluation of Section 8.4. The algorithms presented in this chapter are all implemented in Java, using the Neko framework [UDS02]. The various algorithms are implemented as micro-protocols and composed to form the final protocol stack. Every process in the system runs one of these Neko protocol stacks. Furthermore, all processes are connected pair-wise through TCP channels.

8.5.1 Evaluation environments

Four wide area network environments were used to evaluate the performance of the three atomic broadcast and consensus algorithms. Figure 8.3 shows a schematic representation of these four environments. All machines run a Linux distribution (2.6.8 to 2.6.12 kernels) and a Sun Java 1.5.0 virtual machine. The following paragraphs describe the different wide area network environments in which the atomic broadcast algorithms are evaluated.

Three-location wide area network The first evaluation environment (noted WAN Three Locations, Figure 8.3(a)) is a system with three locations on Grid'5000 [CCD⁺05], a French grid of interconnected clusters designed for the experimental evaluation of distributed and grid computing applications. The round-trip times of the links between the three processes are respectively $2d_0 = 17.2$ ms, $2d_1 = 12.5$ ms and $2d_2 = 10.6$ ms. The observed bandwidth of the three links are respectively 30.1 Mbits/s, 41.4 Mbits/s and 48.7 Mbits/s.

Two-location wide area networks Three environments were used to evaluate the performance of atomic broadcast on wide area networks with two different locations:

- WAN 295 (Figure 8.3(b)): The first two-location environment consists

of one location in Switzerland and one in Japan. The round-trip time between the locations is $2D = 295$ ms and the bandwidth of the connecting link is 1.74 Mb/s.

– WAN 20.1 and WAN 3.9 : The two following environments are systems with both locations on Grid'5000. The WAN 20.1 system (Figure 8.3(c)) features a round-trip time between locations of $2D = 20.1$ ms and a link bandwidth of 32.8 Mb/s. The WAN 3.9 system (Figure 8.3(d)) features a round-trip time between locations of $2D = 3.9$ ms and a link bandwidth of 152 Mb/s.

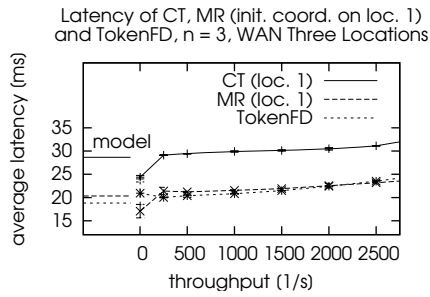
8.5.2 Validation of the model with the experimental results

We now discuss the validation of the model presented in Section 8.4 by the experimental evaluation of the three atomic broadcast algorithms. As mentioned in Section 8.3, the performance graphs present the average latency as a function of the throughput in the system. Furthermore, for the *CT* and *MR* consensus algorithms, the results are given for an initial coordinator that is fixed in one location or shifting with each new consensus execution. The *TokenFD* algorithm has no concept of coordinator and its results are the same for all three settings (they are repeated to give a point of comparison with respect to *CT* and *MR*). As mentioned earlier, all processes *abroadcast* messages at the same rate. The modeled performance of the algorithms is shown on the far-left of each graph (noted “model”). In all performance graphs, the horizontal axis represents the throughput (*i.e.* the global rate at which messages are *abroadcast*) and the vertical axis represents the average latency achieved for a given throughput.

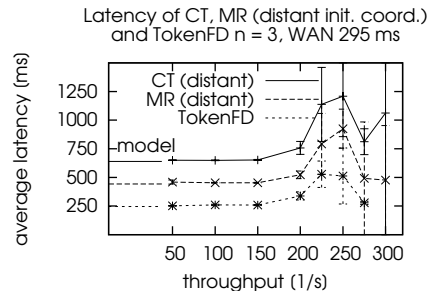
In all experimental setups, the measurements confirm the estimations of the three-location model (Figure 8.4) and in the two-location model (Figures 8.5 to 8.7), especially in the case of moderate throughputs. When the throughput increases, the load on the processors and on the network (which is not modeled) affects the latency of the algorithms (illustrated in particular in Figures 8.5 and 8.6(c)), which increases the gap between the model's estimation and the actual measurements.

Furthermore, when the throughput is very low, as well as in the WAN 3.9 setting, the measured latencies of *CT* and *MR* are lower than what the model predicts. Indeed, our analysis assumes a load in which messages are *abroadcast* often enough that there is always a consensus execution in progress. In the low throughput executions however, there is a pause between the consensus executions. An unordered message that is received during this pause is immediately proposed in a new consensus execution and thus, the *waiting* phase presented in Section 8.4 does not apply to that message. Similarly, in the WAN 3.9 setting, the consensus executions terminate fast enough that the waiting phase for many *abroadcast* messages only

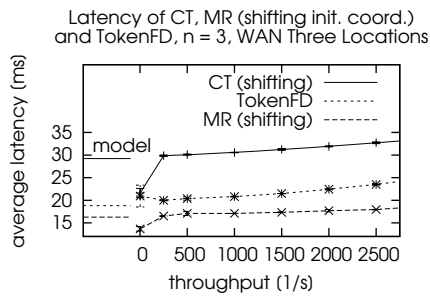
8.5. Experimental performance evaluation



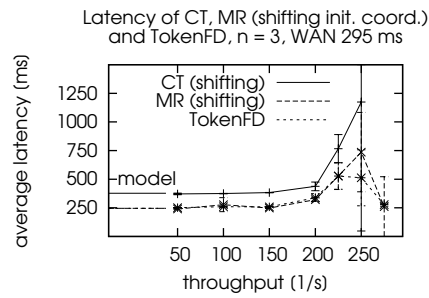
(a) Init. coord. on location 1.



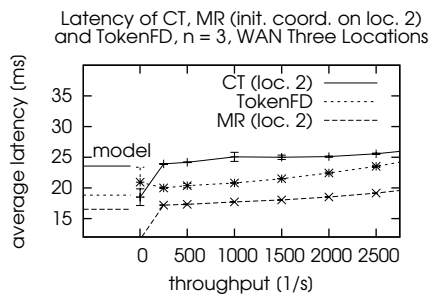
(a) Distant init. coord.



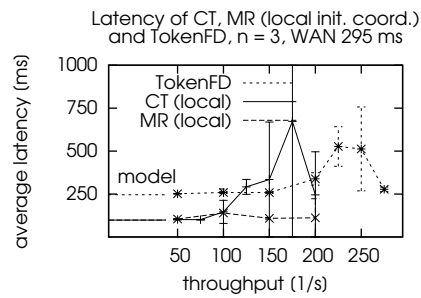
(b) Shifting init. coord.



(b) Shifting init. coord.



(c) Init. coord. on location 2



(c) Local init. coord.

Figure 8.4: Latency vs. throughput of CT, MR and TokenFD in the WAN Three Locations setting.

Figure 8.5: Latency vs. throughput of CT, MR and TokenFD in the WAN 295 setting.

becomes a factor at higher throughputs, where the model is more accurate (Figures 8.7(a) and 8.7(b)).

Finally, the point that was not predicted by the analytical model is the result for high throughputs when the initial coordinator of *CT* and *MR* is on a local location, illustrated by Figures 8.5(c) and 8.6(c). Indeed, in this setting, the system never reaches a stationary state given a sufficiently high throughput. The processes on the local location reach consensus decisions very fast without needing any input from the distant location. The updates that are then sent to the distant location saturate the link between both locations (its bandwidth is only 1.74 Mbits/s in WAN 295 and 32.8 Mbits/s in WAN 20.1). The process on the distant location thus takes decisions slower than the two local processes and prevents the average latency of atomic broadcast from stabilizing. This problem does not affect the settings with a distant or shifting initial coordinator, since the distant location periodically acts as a consensus coordinator, providing a natural flow control. Setup issues, such as the choice of the initial coordinator, thus affect the maximum achievable throughput of the algorithms.

8.5.3 Comparing the performance of the three algorithms

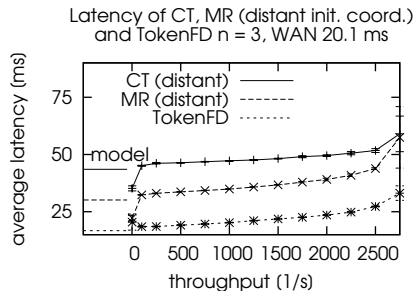
The following section discusses the performance ranking of the three combinations of atomic broadcast and consensus algorithms.

WAN Three Locations The average latency of the three algorithms in the WAN Three Locations environment is presented in Figure 8.4. *TokenFD* and *MR* outperform *CT* for all locations of the initial coordinator and for all throughputs, due to the additional communication step that is needed by the *CT* algorithm. *TokenFD* and *MR* perform similarly when the initial *MR* coordinator is on site 1 (which is the worst-case scenario for *MR*), whereas *MR* achieves slightly better latencies than *TokenFD* for both other initial coordinator locations.

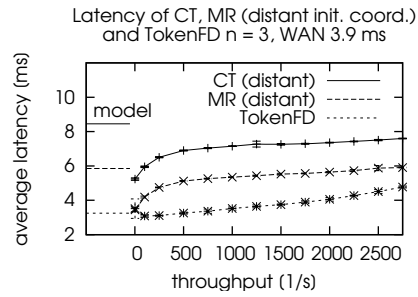
Surprisingly enough, the result of using a shifting initial coordinator in the *CT* and *MR* algorithms are opposite: in the case of *MR*, the latency is lower using a shifting initial coordinator than a fixed initial coordinator on any location, whereas in *CT* it is higher. The explanation is the following: *MR* and *CT* both start a new consensus execution after two communication steps if the coordinator is on a fixed location. If the coordinator shifts, a new execution can start as soon as the next non-coordinator process decides. This is done after *one* communication step in *MR* (if $n = 3$), but after *three* steps in *CT*, as explained in Appendix A.2 and Section 8.4.3.

WAN 295, WAN 20.1 and WAN 3.9 The average latency of the three atomic broadcast and consensus algorithms in the WAN 295, WAN 20.1

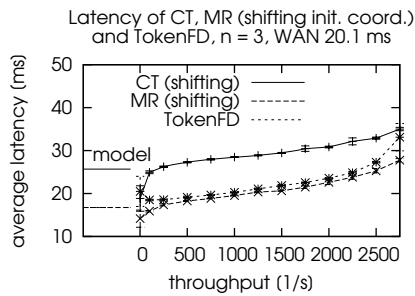
8.5. Experimental performance evaluation



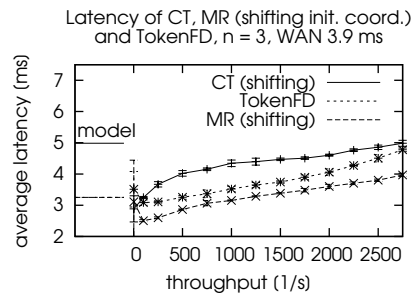
(a) Distant coord.



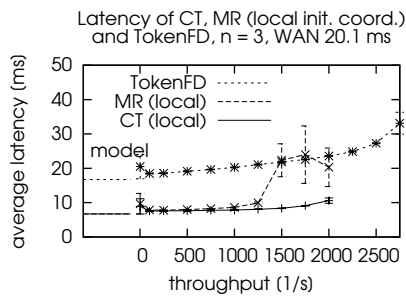
(a) Distant coord.



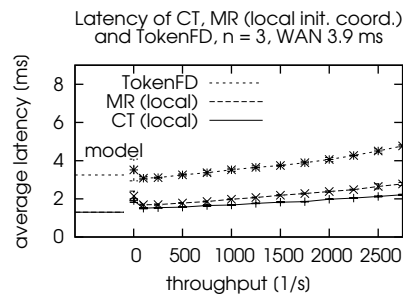
(b) Shifting init. coord.



(b) Shifting init. coord.



(c) Local coord.



(c) Local coord.

Figure 8.6: Latency vs. throughput of CT, MR and TokenFD in the WAN 20.1 setting.

Figure 8.7: Latency vs. throughput of CT, MR and TokenFD in the WAN 3.9 setting.

and WAN 3.9 environments are presented in Figures 8.5 to 8.7. *TokenFD* has lower latencies than *CT* and *MR* when they use a distant initial coordinator (Figures 8.5(a), 8.6(a) and 8.7(a)), whereas the situation is reversed when the coordinator is initially on a local location (Figures 8.5(c), 8.6(c) and 8.7(c)). When the initial coordinator shifts at each new consensus execution, *MR* and *TokenFD* have similar latencies while *CT* is slightly slower. Finally, as mentioned earlier, the low bandwidth of the link between both locations prevents *MR* and *CT* from reaching stable average latencies when the initial coordinator is on the local locations and the throughput is high.

Communication steps versus number of messages: As expected, the performance results presented above show that *communication steps* have the largest impact on performance in wide area networks, whereas the number of sent messages is a key to the performance in a local area network. The validity of this statement however varies with the round-trip time of the network that is considered. As the network latency decreases, the impact of the additional messages that need to be sent and processed increases. In the case of networks with 3.9 ms or even 20.1 ms round-trip times, this impact is clearly observable.

However, for a given set of parameters, the algorithm with the best performance is generally the same (whether we consider a wide area network with a 3.9 ms round-trip time or one with a 295 ms round-trip time) and it is correctly predicted by the model.

Finally, we also saw that choosing a *CT* and *MR* coordinator on the local location (without implementing an additional flow control mechanism) is not necessarily the best solution performance-wise, since the system cannot reach a stationary state as the total throughput increases. Shifting the initial coordinator between locations at each new consensus execution or choosing the *TokenFD* algorithm results in a natural flow control which enables the system to remain in a stationary state even for high throughputs (at the expense of a higher average *adelivery* latency).

8.6 Discussion

The performance of failure detector based atomic broadcast and consensus algorithms has been extensively studied in local area networks, but less so in wide area networks.

In this chapter, we presented a simple analytical model of the performance of three atomic broadcast and consensus algorithms using failure detectors and with different communication patterns. We validated the model with the experimental evaluation of the algorithms in several wide area networks. The evaluation was performed in wide area networks with round-trip times ranging from about 4 to 300 milliseconds to examine the

impact on the trade-off between the number of sent messages and the number of communication steps that the algorithms need to *deliver* a message.

This study confirms that the relative performance between the algorithms is fundamentally different between a local area network and a wide area network (even in wide area networks with small round-trip times): in the former case, the number of sent messages (*i.e.* the number of messages that need to be processed) largely determines the performance of the algorithms, whereas the communication steps have the most impact in the latter case.

Within wide area networks on the other hand, the performance ranking of the three algorithms remains the same, despite the (two order of magnitude) difference in the round-trip time between the smallest and largest wide area networks. Furthermore, this ranking is correctly predicted by our model. The study also showed that algorithms or parameters which provide a natural flow control (such as the *TokenFD* algorithm or the Chandra-Toueg and Mostéfaoui-Raynal consensus algorithms with an initial coordinator that shifts between locations at each new consensus) are effective in reaching higher throughputs in wide area networks.

On the scalability of atomic broadcast algorithms

Several atomic broadcast [ESU04a] and consensus [CT96, MR99] algorithms based on failure detectors have been published. The performance of these algorithms is typically only evaluated in relatively small systems, and the algorithms have been criticized for not scaling well as the number of processes in the system increases. This scalability problem is due to different factors for each of the considered algorithms. The algorithm in [MR99], for example, is affected by the $O(n^2)$ messages that need to be transmitted and processed to solve consensus. In the case of [CT96], a fan-in problem arises: one process needs to receive and handle replies from all other processes in order to solve consensus and thus becomes a bottleneck for the performance of the system. Finally, in the algorithm described in Chapter 4 (and in [ESU04a]), the number of communication steps needed to solve atomic broadcast increases with the size of the system.

It is however desirable for the algorithms to scale beyond a small number of processes: let's take the example of a distributed database on a wide area network with a large number of sites in different geographical locations. The database is replicated across a subset of the sites using active replication with atomic broadcast [Sch93a] and is accessed from all sites on the wide area network. We distinguish between two types of database queries: *read-only* queries that do not modify the database and can be executed on any of the replicas locally (and thus do not need atomic broadcast), and *update* queries that need to be disseminated to all replicas using atomic broadcast. The response time of the database is determined by three factors: (1) the load on the replicas caused by the arrival of queries from the clients, (2) the latency of the atomic broadcast algorithm which increases with the rate of updates to the database and (3) the round-trip time between the client and the replica that handles its request.

If atomic broadcast does not scale beyond a small number of sites, the system can be designed in two ways. Either (i) a small number of sites is

used for replication, in which case the performance of atomic broadcast is acceptable (factor 2), but *all* database queries need to be handled by the same small set of processes. The response time of the distributed database is thus high because all the load is handled by the small set of replicas (factor 1) and because many of the wide area network sites are not in the proximity of one of the (few) replica sites (factor 3). If, on the other hand, (ii) a large number of sites are used for the replication, then the load and proximity problems described above are reduced (factors 1 and 3), since a higher number of replicas handle the same number of client queries as before. However, the high latency of atomic broadcast increases the response time of the distributed database to update queries.

The solution to the problem of the scalability of atomic broadcast (which was informally mentioned in a footnote in Section 4.1.6.B on page 49) is to let a group of kernel processes act as a sequencer: only the kernel processes execute the agreement algorithm and the non-kernel processes are then informed of the outcome of the agreement algorithm. Since the size of the kernel is independent of the size of the system, the number of messages exchanged by the agreement algorithm also becomes independent of the system size. This in turn reduces the cost associated with adding processes to the system.

This approach also has its trade-offs, as the fault tolerance of the entire system is now limited by the fault tolerance of the kernel (but an arbitrary number of non-kernel processes may fail). This trade-off is acceptable, as the size of the kernel (and thus the fault-tolerance of the system) can be tailored to suit the needs of the application and the user.

Surprisingly enough, the performance of this scalable solution to the atomic broadcast problem has not been evaluated (and in particular in systems with a large number of processes). This evaluation is however important: first of all, it confirms that the proposed solution is indeed scalable. Secondly, it justifies limiting the evaluation of consensus and atomic broadcast algorithms to relatively small systems, since in larger systems, the agreement algorithm anyhow only needs to be executed on a subset of all processes.

The results show that there are *several* factors that limit the scalability of the considered algorithms, but that by limiting the ordering algorithm to a kernel of processes, the algorithm indeed becomes scalable and its performance degrades gracefully as the size of the system increases. Furthermore, when the system is small, the latency of the scalable atomic broadcast algorithm is close to the latency of the underlying atomic broadcast algorithm it depends on.

Related work In [RGS98], the authors present a scalable atomic multicast algorithm. The scope of their work is however different from ours,

since they focus on atomic *multicast* (*i.e.* a total order broadcast to a subset of all processes in the system). Their primary concern is to avoid having to broadcast a message to all processes in the system if only a subset of these processes are part of the destination of the message. The algorithm presented in [RGS98] minimizes the number of sent messages for each atomic multicast, but does so at the expense of the latency of delivery (6 communication steps are necessary to atomically deliver a message, where other algorithms such as [CT96] need 4 steps).

In [Urb03], the author studies the scalability of the consensus algorithms presented in [CT96] and [MR99]. However, the study is performed in a *simulated* environment and assumes that the network is an exclusive resource (*i.e.* two processes cannot send a message on the network at the same time). As a consequence, the results presented in the study are not representative of the performance of the algorithms in a real execution environment. Furthermore, no solution to the scalability problem is proposed or studied in [Urb03].

Another study that specifically focuses on atomic broadcast in large scale systems is carried out in [RFV96]. The authors propose an atomic broadcast algorithm that is a hybrid between a token based and a fully decentralized approach (and that relies on an underlying group membership service). Once again however, the evaluation of this algorithm is done by simulation.

The Chubby distributed lock service [Bur06] implements some of the techniques that are also used by the scalable atomic broadcast in this chapter. Also, the kernel and non-kernel processes of the scalable atomic broadcast algorithm correspond to, respectively, *acceptor* and *learner* agents in the Paxos consensus algorithm [Lam98].

Finally, in [BK02], the authors examine the performance of distributed algorithms in a wide area network of 11 hosts. The goal of [BK02] is however not to study the scalability of the different algorithms, but to evaluate the effect of network latency and network loss on their performance. Moreover, the constraints on the message flow of the algorithms considered in [BK02] are not applicable to the algorithms that are studied in this chapter.

Contributions A formal presentation of the scalable atomic broadcast algorithm is given in this chapter, as well as a sketch of its correctness. We also discuss under which conditions it is interesting to use the scalable atomic broadcast algorithm instead of the underlying atomic broadcast algorithm. The major contribution of this chapter is then the performance evaluation of the scalable algorithm (applied to three underlying atomic broadcast and consensus implementations), which shows that limiting the agreement protocol to a kernel of processes is indeed a solution to the scal-

ability of atomic broadcast algorithms based on failure detectors.

The performance of the scalable atomic broadcast algorithm is evaluated in two settings: (1) on a local area network and (2) on a wide area network of seven interconnected clusters. The size of the system varies from 3 to 23 processes and two kernel sizes are considered: 3 processes (supporting a single failure) and 7 processes (supporting two or three failures, depending on the underlying atomic broadcast algorithm). In both settings, the performance of the scalable atomic broadcast algorithm is compared to the original atomic broadcast algorithm.

The structure of the chapter is the following. In Section 9.1, we present the system model, as well as the atomic broadcast and consensus algorithms that are considered later. The scalable atomic broadcast algorithm is presented in Section 9.2 with a discussion on its benefits as well as a sketch of its proof and the optimizations that were applied. Section 9.3 starts by detailing the setup of the performance evaluation experiments and the various parameters that affect the measurements. The results of the measurements in the local and wide area networks are then presented and analyzed. Finally, Section 9.4 concludes the chapter.

9.1 System model

9.1.1 System model, consensus and atomic broadcast

We consider an asynchronous system of n processes (taken from a set Π) augmented with failure detectors (see Chapter 3 for additional details). Among the n processes in Π , a subset of size k , called the *kernel*, execute the agreement algorithm. If all processes execute the agreement algorithm, then $kernel = \Pi$.

The processes communicate by message exchange over quasi-reliable channels and at most f processes among the k kernel processes may fail by crashing (*i.e.* no Byzantine faults). A process is *correct* if it never crashes and faulty otherwise.

The three following agreement problems are further discussed in this chapter and are therefore briefly presented here:

9.1.1.A Reliable multicast

Informally, in the reliable multicast problem, a set of processes need to agree on a common set of delivered messages. Each message m has a tag $group(m)$ indicating the set of destination processes. The reliable multicast problem is defined by two primitives $rmulticast$ and $rdeliver$ that satisfy the three following properties [HT94]:

Validity If a correct process *rmulticasts* a message m , then some correct process in $group(m)$ eventually delivers m or no process in that group is correct.

Agreement If a correct process *rdelivers* a message m , then all correct processes in $group(m)$ eventually *rdeliver* m .

Uniform integrity For any message m , every process *rdelivers* m at most once and only if p is in $group(m)$ and m was previously *rmulticast*.

If the destination group $group(m)$ is equal to Π for all messages m , then Reliable multicast reduces to Reliable broadcast [HT94]. In this chapter, we consider a reliable multicast algorithm similar to the reliable broadcast algorithm presented in [CT96]: for each *rmulticast* message m , whenever a process in $group(m)$ receives m for the first time, it *rdelivers* m and sends m to all processes in $group(m)$. For a group $group(m)$ of size n_g , this algorithm requires $O(n_g^2)$ messages and a single communication step to *rdeliver* a message m .

Reliable *multicast* is introduced here since the system is divided into two sets of processes: non-kernel and kernel processes. To reliably send messages only to a subset of the system, reliable multicast is needed, rather than reliable broadcast (which sends messages to *all* processes).

9.1.1.B Consensus

Informally, in the consensus problem, a group of processes have to reach a common decision. The formal specification of the consensus problem is presented in Section 3.2.1.

In this chapter, we consider two consensus algorithms that use the $\diamond S$ failure detector: (1) the Chandra-Toueg algorithm presented in [CT96] (noted *CT*) and (2) the Mostéfaoui-Raynal algorithm presented in [MR99] (noted *MR*).

The *CT* consensus algorithm is a centralized algorithm that reaches a decision using $O(n)$ messages and one reliable broadcast. It requires 2 to 3 communication steps to reach a decision in runs without failures and without wrong suspicions (*good* runs). A detailed description of the algorithm is presented in Appendix A.2.1.

The *MR* consensus algorithm is a decentralized algorithm that requires at least 2 communication steps and $O(n^2)$ messages for all processes to reach a decision. The algorithm is described in detail in Appendix A.2.2.

9.1.1.C Atomic broadcast

In the atomic broadcast problem, a set of processes have to agree on a common order of delivery of a set of messages. It is a generalization of the

reliable broadcast problem with an additional ordering constraint. The formal definition of atomic broadcast is presented in Section 3.2.3.

The scalable atomic broadcast algorithm uses an underlying atomic broadcast algorithm. Two such atomic broadcast algorithms are considered in this chapter: the Chandra-Toueg atomic broadcast algorithm [CT96] (presented in detail in Appendix A.3.1) and the token based atomic broadcast algorithm using an unreliable failure detector presented in Section 4 (noted *TokenFD*).

9.2 Scalable atomic broadcast

9.2.1 Presentation of the algorithm

Algorithm 9.1: Scalable atomic broadcast algorithm (code of process p)

```

1: Initialisation:
2:    $kernel \leftarrow$  a subset of  $\Pi$ 
3:    $i \leftarrow 1$  {serial number for adeliver }
4:   if  $p \in kernel$  then
5:      $originalAbcast \leftarrow$  the underlying atomic broadcast implementation

6:   procedure  $abroadcast(m)$  {To abroadcast a message  $m$ }
7:     if  $p \in kernel$  then
8:        $originalAbcast.abroadcast(m)$  {Order message  $m$ }
9:     else
10:       $send(m, abcast)$  to  $kernel$  {send abroadcast request to kernel}

11:  when receive  $(m, abcast)$  {abroadcast request from non-kernel process}
12:     $originalAbcast.abroadcast(m)$ 

13:  when  $originalAbcast.adeliver(m)$  {only if  $p \in kernel$ }
14:    if  $m$  is  $A$ -delivered for the first time by  $originalAbcast$  then
15:       $adeliver(m)$ 
16:       $send(m, adeliver, i)$  to  $\Pi - kernel$ 
17:       $i \leftarrow i + 1$ 

18:  when receive  $(m, adeliver, i)$  {only if  $p \in \Pi - kernel$ }
19:     $adeliver(m)$ 
20:     $i \leftarrow i + 1$ 

```

Algorithm 9.1 presents the code of the scalable atomic broadcast algorithm. To simplify the presentation of the algorithm, optimization issues

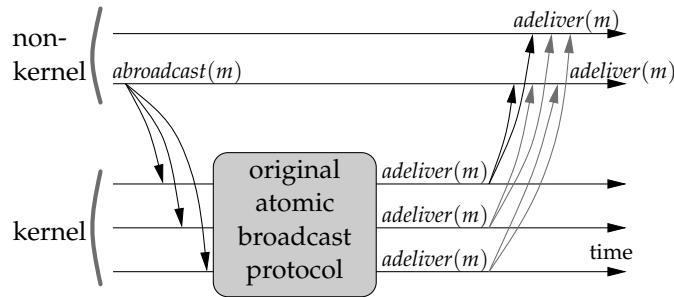


Figure 9.1: Execution of the scalable atomic broadcast algorithm: the kernel processes act as a sequencer for all messages sent in the system

(some of which depend on the underlying atomic broadcast algorithm) are discussed later in Section 9.2.4.

To *abroadcast* a message m , a kernel process calls the *abroadcast* primitive of the underlying atomic broadcast algorithm (line 8), whereas a non-kernel process sends an *abroadcast* request to all kernel processes (line 10).

Whenever a kernel process receives an *abroadcast* request from a non-kernel process (line 11), it *abroadcasts* the message using the underlying atomic broadcast implementation (line 12). Notice that this can lead to several *abroadcast* invocations on the underlying atomic broadcast algorithm for a single *abroadcast* request from a non-kernel process. This is discussed in Section 9.2.4.A.

Whenever the underlying atomic broadcast implementation *adelivers* a message m for the first time (which can only happen on the kernel processes, lines 13 and 14), m is *adelivered* (line 15) and forwarded to all non-kernel processes (line 16). Again, this can lead to several (duplicate) transmissions for a single *adeliver*. This is again discussed in Section 9.2.4.B.

Finally, whenever a non-kernel process receives *adeliver* information for a message m from a kernel process (line 18), m is *adelivered*. The counter i ensures that the order of *adeliveries* is respected and that no message is *adelivered* twice.

The algorithm behaves similarly to sequencer algorithms, where messages that need to be ordered are sent to a sequencer that assigns a sequence number to the messages. Here, instead of having a single sequencer process, a group of processes (the *kernel*) acts as a sequencer. This is illustrated in Figure 9.1: a non-kernel process p *abroadcasts* a message and sends it to be sequenced by the kernel. The kernel processes execute the atomic broadcast algorithm and send its result to all non-kernel processes. Eventually, all correct processes *adeliver* the message.

Finally, if the *kernel* contains all processes (*i.e.* all processes participate in the agreement algorithm), Algorithm 9.1 reduces to the underlying atomic broadcast algorithm.

9.2.2 Benefits and drawbacks of the scalable algorithm

Algorithm 9.1 transforms an atomic broadcast algorithm into a sequencer-like algorithm that is well-suited for systems with a large number of processes. In this section, we discuss the benefits on time and message complexity of using the scalable algorithm instead of the underlying atomic broadcast algorithm in large systems. Finally, we also present the drawbacks of the scalable atomic broadcast on the resilience in large systems.

9.2.2.A Impact on message complexity

First of all, we examine the impact of the scalable atomic broadcast algorithm on the message complexity, *i.e.* the number of messages exchanged by the processes. We assume that the underlying atomic broadcast algorithm exchanges $c_{msg}(n)$ messages per atomic broadcast, where n is the size of the system and c_{msg} a function of n . The Chandra-Toueg atomic broadcast algorithm (coupled with the *CT* consensus algorithm and the reliable broadcast algorithm in [CT96]) needs $c_{msg}(n) = 2n^2 + 2n (= n^2 + n + n + n^2)$ messages per atomic broadcast.

With Algorithm 9.1 (and the optimizations discussed in Section 9.2.4 below) and a kernel of k processes, $c_{msg}(k) + (n - k)$ messages are needed per atomic broadcast: $c_{msg}(k)$ to order the message with the underlying atomic broadcast implementation and $(n - k)$ messages to notify the non-kernel processes. The scalable atomic broadcast thus reduces the number of messages that are needed from $c_{msg}(n)$ to $c_{msg}(k) + (n - k)$.

If n is large, the scalable atomic broadcast algorithm is interesting as soon as the number of messages $c_{msg}(n)$ needed by the underlying atomic broadcast algorithm is superlinear in n . Indeed, assume that $O(c_{msg}(n)) > O(n)$. In this case, we have $O(c_{msg}(k) + (n - k)) = O(n - k) = O(n) < O(c_{msg}(n))$ (since $c_{msg}(k)$ is independent of n , we have $O(c_{msg}(k)) = O(1)$). For example, in the case where the Chandra-Toueg atomic broadcast (coupled with the reliable broadcast algorithm in [CT96] and *CT* or *MR* consensus) is used as the underlying implementation, we have $O(c_{msg}(n)) = O(n^2)$ and thus, the scalable atomic broadcast needs significantly less messages as the system size increases. In the case of the *TokenFD* algorithm, we have $O(c_{msg}(n)) = O(n)$ and the scalable algorithm does not significantly reduce the number of sent messages as the system size increases.

9.2.2.B Impact on time complexity

We now examine how the scalable atomic affects the time complexity, *i.e.* the number of communication steps needed to *adeli*ver messages, as the size of the system increases. We assume that the underlying atomic broadcast algorithm needs at least $c_{time}(n)$ communication steps for all processes to

adeliver a message, where n is the size of the system and c_{time} is a function on n .

With Algorithm 9.1 and the optimizations from Section 9.2.4, the non-kernel processes *adeliver* a message m one communication step after the kernel process responsible of sending notifications *adelivers* m . This kernel process is the coordinator in the case of *CT* or *MR* consensus (used by the Chandra-Toueg atomic broadcast algorithm) or the token holder in the case of *TokenFD*. In the case of the *TokenFD* and the *CT* algorithms, the non-kernel processes *adeliver* a message m at the same time as the last kernel processes and thus the scalable algorithm needs $c_{time}(k)$ communication steps to *adeliver* m . In the case of *MR*, all kernel processes *adeliver* a message after three communication steps (reliable broadcast takes 1 step, consensus 2 steps). The non-kernel processes *adeliver* one communication step later, *i.e.* after $c_{time}(k) + 1$ steps.

In the case of *CT* and *MR*, we have respectively $c_{time}(n) = 3$ and $c_{time}(n) = 2$, which are both independent of the system size. In the case of *TokenFD* however, we have $c_{time}(n) = 2 + f$, with $f \cdot (f + 1) + 1 \leq n$, and thus $O(c_{time}(n)) = O(\sqrt{n})$. In this case, the scalable atomic broadcast algorithm allows messages to be delivered after a constant number of communication steps, instead of a number of steps that increases with n .

9.2.2.C Impact on resilience

The resilience of Algorithm 9.1 is equal to the resilience of the underlying atomic broadcast algorithm executed on the kernel processes. The resilience of the scalable algorithm in a system with n processes (among which k are kernel processes) is thus *lower* than the resilience of the underlying atomic broadcast algorithm if it is executed on all n processes.

At first hand, this seems to be a disadvantage of the scalable algorithm. However, the scalable atomic broadcast algorithm allows a *compromise* between the performance and the resilience of a system. By tailoring the kernel size to match the fault tolerance requirements of the system, higher performance is reached with Algorithm 9.1 than with the underlying atomic broadcast algorithm (which has a resilience that is *higher* than the requirements of the system). If the kernel includes all processes, then the scalable atomic broadcast reduces to the underlying atomic broadcast implementation and (almost) no performance penalty is incurred.

9.2.3 Proof of correctness

The following paragraphs present the proof of correctness of the scalable atomic broadcast algorithm. We assume that k processes are in the kernel ($k \leq n$). The proof of *Uniform integrity* is easy and not shown here.

Validity. If a correct process p *abroadcasts* m , it either (1) *abroadcasts* m using the underlying atomic broadcast algorithm (kernel process, line 8) or (2) sends an *abroadcast* request to the kernel (non-kernel process, line 10). From the properties of quasi-reliable channels and lines 11 and 12, case (2) reduces to case (1). From the *Validity* property of the underlying atomic broadcast algorithm, all correct processes in the kernel eventually *adeliver* m (line 13) and send a message to the non-kernel processes indicating that m was *adelivered*. Since at least one kernel process is correct, all correct non-kernel processes eventually receive the message (line 18) and *adeliver* m (line 19). Process p thus eventually *adelivers* m . \square

Uniform agreement. If a process p *adelivers* m , it either (1) does so at line 15 (if it's a kernel process) or (2) at line 19 (if it's a non-kernel process). Case (2) reduces to case (1): if p *adelivers* m , it has received $(m, \text{adeliver}, i)$ from a kernel process. From the *Uniform agreement* property of the underlying atomic broadcast algorithm, all correct kernel processes eventually *adeliver* m (line 15). Finally, from the properties of quasi-reliable channels, all correct non-kernel processes eventually receive the update concerning m and *adeliver* m . \square

Uniform total order. Assume for contradiction that process p *adelivers* m before m' whereas process q *adelivers* m' before m . If p and q are both kernel processes, this leads to a contradiction, following the *Uniform total order* of the underlying atomic broadcast algorithm. If p is a kernel process and q a non-kernel process, then p sent $(m, \text{adeliver}, i)$ and $(m', \text{adeliver}, i')$ with $i < i'$. On the other hand, q received $(m, \text{adeliver}, i)$ and $(m', \text{adeliver}, i')$ with $i > i'$ (and the messages were sent by a kernel process). From the *Uniform total order* of the underlying atomic broadcast algorithm, this leads to a contradiction, since all kernel processes *adeliver* the messages in the same order (and send the updates with the same values of the counter i). The reasoning also applies to the case where p and q are both non-kernel processes. \square

9.2.4 Optimizations

Algorithm 9.1 presented a scalable atomic broadcast algorithm that can be combined with any underlying atomic broadcast algorithm. Consequently, the scalable algorithm contains some inefficiencies that can be avoided by knowing which underlying atomic broadcast implementation is used. The three main optimizations that were applied to the scalable algorithm are now described.

9.2.4.A Avoiding unnecessary *abroadcasts*

In Algorithm 9.1, each time a non-kernel process *abroadcasts* a message, it sends a request to *all* kernel processes. These kernel processes then invoke the *abroadcast* primitive of the underlying atomic broadcast implementation. If k processes are part of the kernel, this results in $k - 1$ unnecessary invocations of *abroadcast*. This impacts the performance of the algorithm, since these $k - 1$ additional messages need to be handled, ordered and delivered (and finally discarded).

The number of unnecessary *abroadcasts* can be limited by taking advantage of the underlying atomic broadcast implementation. For example, in the Chandra-Toueg atomic broadcast algorithm, whenever a message is *abroadcast*, it is first reliably broadcast to all group members. Following Algorithm 9.1, a non-kernel process that *abroadcasts* a message m would thus first send a request containing m to the kernel processes, which would then (k times) reliably broadcast m to all kernel processes in the underlying Chandra-Toueg algorithm. To avoid the $k - 1$ unnecessary reliable broadcasts, the non-kernel process could, upon *abroadcasting* m , instead *directly* reliably broadcast message m to all kernel processes (instead of sending the request message that results in k reliable broadcasts). This optimization ensures that the cost of each *abroadcast* message remains the same as the cost of a single *abroadcast* invocation of the underlying atomic broadcast implementation.

When the *TokenFD* algorithm is used as the underlying atomic broadcast and if a non-kernel process *abroadcasts* a message m , it simply needs to send m to all kernel processes (as opposed to reliably broadcasting m , if the Chandra-Toueg atomic broadcast algorithm is used).

9.2.4.B Avoiding redundant *adeliver* notifications

In Algorithm 9.1, all k kernel processes send *adeliver* notifications to the $n - k$ non-kernel processes (line 16). Out of these $k \cdot (n - k)$ notifications, $(k - 1) \cdot (n - k)$ are redundant. The optimization consists in letting a single kernel process send the *adeliver* notification, instead of all k kernel processes. For example, if the Chandra-Toueg algorithm with *CT* consensus is used as the underlying atomic broadcast implementation, then the first kernel process to *adeliver* a message m is the coordinator of the consensus execution where m 's order was decided. To minimize the time between a consensus decision and its dissemination to non-kernel processes, only the coordinator process sends the message containing the consensus decision to all non-kernel processes.

In the *TokenFD* algorithm, the token-holder that *adelivers* a message for the first time is responsible for notifying the non-kernel processes, whereas the *CT* or *MR* consensus coordinator takes this role if the Chandra-Toueg

algorithm is the underlying atomic broadcast implementation.

This solution can lead to lost or duplicate decision messages in runs with failures or suspicions. Duplicate decision messages are discarded based on their identifier. A simple retransmission mechanism solves the problem of lost messages: if a non-kernel process detects that it has missed an *adeliver* notification, it sends a retransmission request to the kernel processes. All correct kernel processes then reply by sending the missing notification.

9.2.4.C Sending notifications in batches

In the scalable atomic broadcast algorithm presented above, whenever a message is *adelivered*, it is sent to all non-kernel processes (lines 13 to 17, Algorithm 9.1). Thus, even if the underlying atomic broadcast algorithm orders messages in batches, the delivery notifications for these messages are all sent individually to the non-kernel processes. These notifications can easily be sent as a single batch if we know the underlying atomic broadcast implementation, as was the case for avoiding unnecessary *abroadcasts* in Section 9.2.4.A above.

This optimization applies to both the Chandra-Toueg and the *TokenFD* atomic broadcast algorithm, as both algorithms order messages in batches.

9.3 Performance evaluation

9.3.1 Performance metrics, workload and the implementation framework

The following paragraphs describe the benchmarks (*i.e.* the performance metrics and the workloads) that were used to evaluate the performance of the scalable atomic broadcast algorithm. Similar benchmarks have been presented in Chapters 7 and 8, as well as in [Urb03, USS03, ESU04a]. The framework in which the algorithm was implemented is detailed at the end of this section.

9.3.1.A Performance metric: latency vs. throughput

The performance metric that was used to evaluate the algorithms is the latency of atomic broadcast. For a single atomic broadcast, the latency L is defined as follows. Let t_a be the time at which the *abroadcast*(m) event occurred and let t_i be the time at which *adeliver*(m) occurred on process p_i , with $i \in 0, \dots, n - 1$. The latency L is then defined as $L \stackrel{\text{def}}{=} (\frac{1}{n} \sum_{i=0}^{n-1} t_i) - t_a$. In our performance evaluation, the mean for L is computed over many

messages and for several executions. 95% confidence intervals are shown for all the results.

As we already discussed in Section 7.2.1 on page 115, other metrics such as the *early* latency could have been used. The early latency metric however has a bias towards low *adelivery* times, which, in the scalable atomic broadcast algorithm, always occur on kernel processes. The early latency metric doesn't take into account the cost of diffusing the results of the agreement protocol among kernel processes to non-kernel processes and is therefore not suitable here.

9.3.1.B Workloads

The latency L is measured for a certain workload, which specifies how the *abroadcast* events are generated. We chose a simple symmetric workload where all processes send atomic broadcast messages¹ at the same constant rate and the *abroadcast* events come from a Poisson stochastic process. The global rate of atomic broadcasts is called the *throughput* T , which is expressed in messages per second (or msgs/s). We then evaluate the dependency between the latency L and the throughput T .

Furthermore, we only consider the system in a stationary state, when the rate of *abroadcast* messages is equal to the rate of *adelivered* messages. This state can only be reached if the throughput is below some maximum threshold T_{max} . Beyond T_{max} , some processes are left behind. We ensure that the system stays in a stationary state by verifying that the latencies of all processes stabilize over time.

Finally, we only evaluate the performance of the algorithms in good runs, *i.e.* without any process failures or wrong suspicions. The latency of the algorithms is measured once the system has reached a stationary state (at a sufficiently long time after the startup). The parameters that influence the latency are n (the number of processes), the algorithm (scalable or non-scalable atomic broadcast), the throughput and, in the case of the scalable algorithm, k (the number of kernel processes).

9.3.1.C Implementation framework and issues

The scalable atomic broadcast algorithm was implemented in Java, using the Neko framework [UDS02]. In this framework, the various algorithms are implemented as microprotocols. These microprotocols are then composed together to form the final protocol stack. Every process in the system runs one of these Neko protocol stacks. Furthermore, all processes are

¹The atomic broadcast messages do not contain any payload, in order to reach the maximum possible performance when comparing the original and the scalable atomic broadcast algorithms

connected pair-wise through TCP channels²

The atomic broadcast algorithms already implemented in Neko (that have been used for the performance evaluations in the previous chapters and in [USS03, UHSK04, CUBS02, Urb03]) were reused as the underlying atomic broadcast algorithm for the scalable atomic broadcast algorithm. Furthermore, the scalable atomic broadcast algorithm was implemented with all the optimizations described in Section 9.2.4.

9.3.2 Evaluation environment

The following paragraphs present the details of the two experimental setups that were used to evaluate the scalable atomic broadcast algorithm. We started by measuring the performance of the algorithm on a local area network of interconnected machines. Then, the same experiments were repeated on a wide area network of interconnected clusters of machines to evaluate if the scalability properties of the algorithm hold even if the kernel processes are on different sites.

9.3.2.A Local Area Network

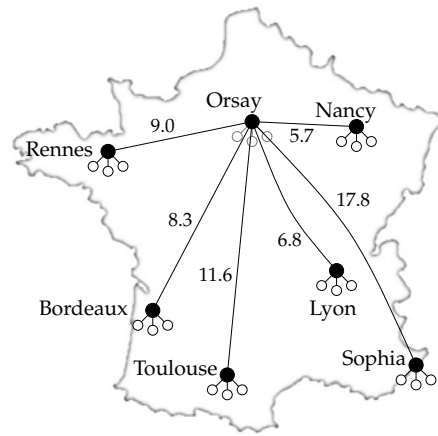
The first set of experiments were executed on the local area network cluster previously presented in Chapter 7. Each node in the local area network has a Pentium 4 processor, model number 630, at 3 GHz and with 2 MB of L2 cache. The size of the main memory on each node is 1 GB and the nodes are interconnected by a single Gigabit Ethernet switch. The round-trip time between two nodes is approximately 0.1 ms. All nodes run a SuSE Linux distribution (with a 2.6.11 kernel) and Sun's Java 1.5.0_05 64-bit Server Virtual Machine.

9.3.2.B Wide Area Network: Grid'5000

The second set of experiments were executed on Grid'5000, which was previously used in Chapter 8.

Grid'5000 is composed of 14 sites (in 9 geographical locations), among which seven were used for our measurements: Bordeaux, Lyon-capricorne, Nancy, Orsay, Rennes-parasol, Sophia and Toulouse. In order to examine the influence of the round-trip times on the scalable atomic broadcast algorithm, we chose to use 7 sites (in 7 different locations) to ensure that all kernel processes would always be on separate sites. When more than

²We also tested the algorithms with a simple UDP and IP multicast implementation of reliable channels. The results were however comparable to the TCP implementation (or worse than TCP, for high throughputs without flow control), which suggests that in our setup the main cost of sending a multicast message is its serialization and processing, rather than the number of copies that are actually transited on the network hardware.



(a) Geographical locations of the sites

	Bordeaux	Lyon	Nancy	Orsay	Rennes	Sophia
Lyon	10.5					
Nancy	12.6	10.6				
Orsay	8.3	6.8	5.7			
Rennes	8.0	12.6	13.3	9.0		
Sophia	10.6	7.2	17.2	17.8	19.2	
Toulouse	5.7	8.8	18.7	11.6	20.7	15.3

(b) Round trip times (*ms*) between sites

Figure 9.2: Geographical distribution of the sites in the Grid'5000 setup with their round trip times to the Orsay site (9.2(a)) and the round trip times between all sites (9.2(b))

seven processes participate in a given experiment, the additional processes are distributed evenly among the seven sites.

The following paragraphs give some additional details on the hardware on the different sites and the latencies between sites.

The hardware and software setup of the nodes on the different sites are the following:

Bordeaux, Rennes-parasol, Toulouse Dual-processor nodes with AMD Opteron 248 processors at 2.2 GHz and with 1MB of L2 cache. All nodes have 2GB of main memory and are interconnected by Gigabit Ethernet. Sun's Java 1.5.0_06 AMD64 Server Virtual Machine was installed on the nodes.

Lyon-capricorne, Nancy, Orsay, Sophia Dual-processor nodes with AMD

Opteron 246 processors at 2 GHz and with 1MB of L2 cache. All nodes have 2GB of main memory and are interconnected by Gigabit Ethernet. Sun's Java 1.5.0_06 AMD64 Server Virtual Machine was installed on the nodes.

All hosts run a Debian Linux distribution (with a 2.6.8 or a 2.6.12 kernel).

The coordinator process of the *CT* and *MR* consensus algorithms always runs on a host of the Orsay site. Figure 9.2(a) shows the geographical distribution of the seven sites, as well as an estimate of the round-trip times (in milliseconds) between all sites and the Orsay site. The *TokenFD* passes the token among sites in the following order: Orsay – Bordeaux – Nancy – Lyon – Toulouse – Sophia – Rennes. The (symmetrical) matrix of the round trip times between sites is shown in Figure 9.2(b).

9.3.3 Results of the performance measurements

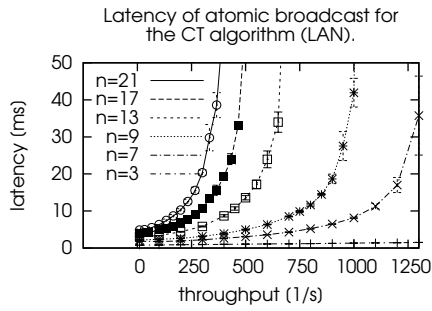
The following paragraphs present the results of the performance measurements that were obtained using the two setups previously described.

9.3.3.A Local Area Network

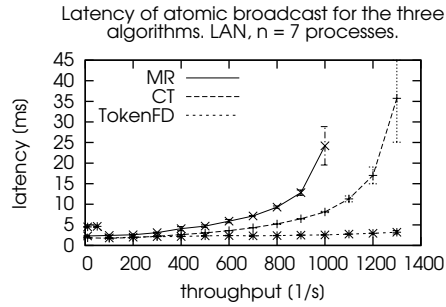
The original algorithm: Figure 9.3 shows the latency versus the throughput of the Chandra-Toueg (Figures 9.3(a) and 9.3(b)) and *TokenFD* (Figure 9.3(c)) atomic broadcast algorithms when all processes in the system participate in the algorithms. The figure shows results for system sizes ranging from 3 to 21 processes, supporting from 1 to 10 failures (*CT* and *MR*) or 1 to 4 failures (*TokenFD*). The horizontal axis shows the load on the system (in messages per seconds), whereas the average latency is shown on the vertical axis (in milliseconds).

One clearly sees that as the number of processes increases, the original atomic broadcast algorithms do not scale well. In the case of a small system with 3 processes, the latency remains almost constant as the throughput increases, for all three algorithms. In the largest system with 21 processes, however, the latency increases extremely fast with the throughput, especially for the *CT* and *MR* algorithms, where $O(n^2)$ messages are needed to solve atomic broadcast. Furthermore, the throughput in a system with 21 processes is limited to 300 and 400 messages per second for *MR* and *CT* respectively. In the case of *TokenFD*, the scalability problem is less severe (since only $O(n)$ messages are sent by the algorithm) but the latency still increases with the system size, since $O(\sqrt{n})$ communication steps are needed to *deliver* messages.

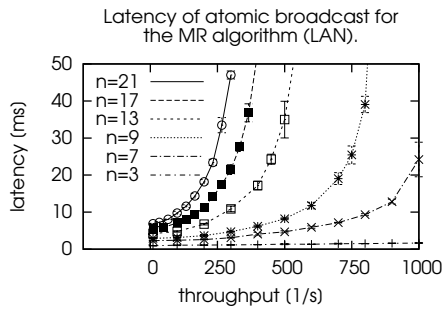
Finally, the difference between the *CT* and *MR* algorithms requiring $O(n^2)$ messages and the *TokenFD* algorithm requiring $O(n)$ messages is il-



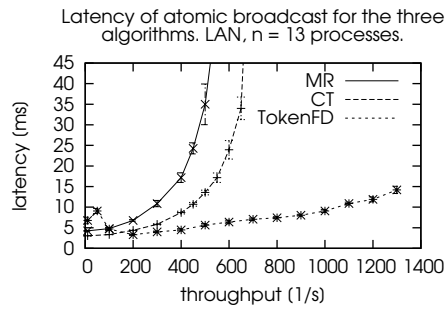
(a) CT



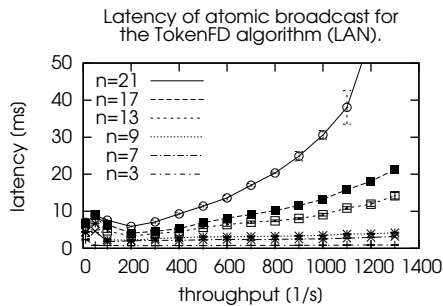
(a) n = 7 processes



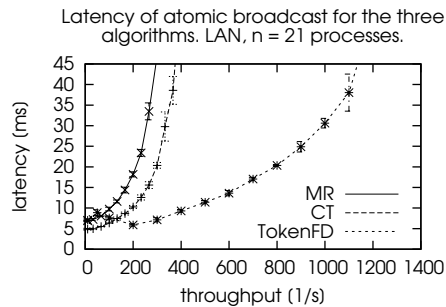
(b) MR



(b) n = 13 processes



(c) TokenFD



(c) n = 21 processes

Figure 9.3: Latency vs. throughput of the Chandra-Toueg (with CT or MR consensus) and TokenFD atomic broadcast algorithms in a local area network

Figure 9.4: Latency vs. throughput of the three atomic broadcast algorithms for system sizes of 7, 13 and 21 processes in a local area network.

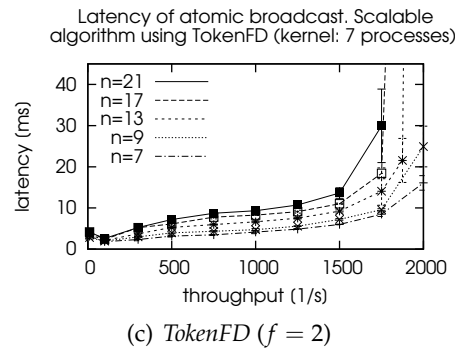
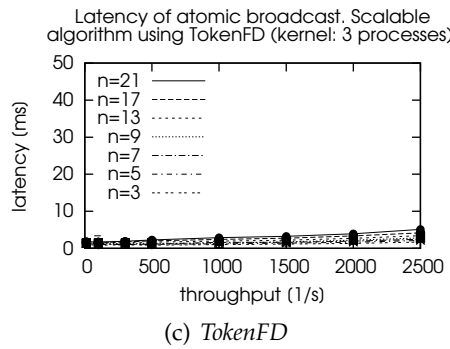
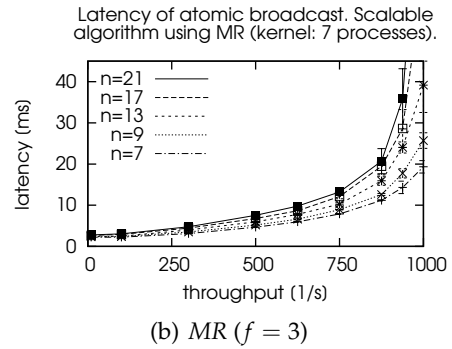
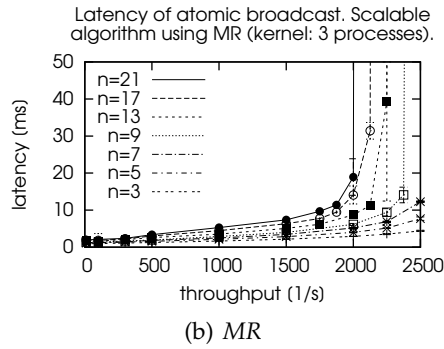
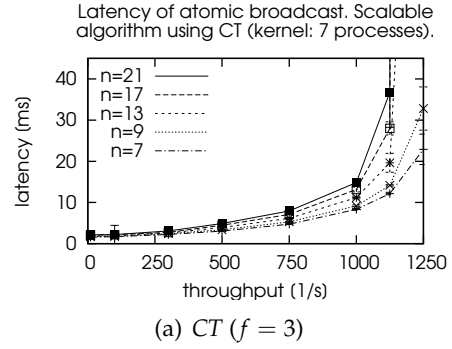
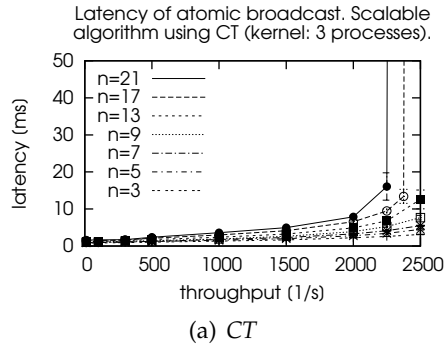


Figure 9.5: Latency vs. throughput of the scalable atomic broadcast algorithm with a kernel of 3 processes in the local area network setup.

Figure 9.6: Latency vs. throughput of the scalable atomic broadcast algorithm with a kernel of 7 processes in the local area network setup.

illustrated by Figure 9.4. In the case of a small system of $n = 7$ processes (Figure 9.4(a)), all three algorithms achieve relatively low latencies, even as the throughput increases. On the other hand, when 21 processes form the system, only *TokenFD* reaches throughputs larger than 400 msgs/s (Figure 9.4(c)).

The scalable algorithm: Figures 9.5 and 9.6 show the latency versus the throughput of the scalable atomic broadcast algorithm which executes the agreement algorithm only on the kernel, a subset of all processes. Two kernel sizes, $k = 3$ (supporting 1 failure) and $k = 7$ (supporting 3 (*CT*, *MR*) or 2 (*TokenFD*) failures) were considered. This time, the dependency between latency and throughput is similar for all system sizes. In the case of a kernel of three processes (Figure 9.5), the latency only slightly increases with the throughput, for all system sizes that were considered. For *CT* and *MR*, there is a threshold (around 2000 msgs/s, Figures 9.5(a) and 9.5(b)) above which the latency quickly degrades, especially for large systems. For the *TokenFD* algorithm, this threshold was not met and is above 2500 messages per second.

In the case of a kernel with seven processes (Figure 9.6), the performance of the system is once again strongly correlated to the performance of the underlying atomic broadcast algorithm running on the kernel (with a slight overhead due to the diffusion of updates to the non-kernel processes). Again, high throughputs (around 1100 msgs/s) are reached by the scalable algorithm, whether *CT*, *MR* or *TokenFD* is used as the underlying atomic broadcast algorithm.

The measurements with a kernel of 7 processes confirm the results observed for a kernel of 3 processes. Indeed, the dependency between latency and throughput is relatively similar for all system sizes and is mainly influenced by the size of the kernel (the results in Figure 9.6 are similar to the case $n = 7$ in Figure 9.3).

Finally, Figure 9.7 shows the latency of atomic broadcast as a function of the number of processes in the system, for two fixed throughputs and for the three algorithms *CT*, *MR* and *TokenFD*. For *CT* and *MR*, two trends are visible in this figure: on one hand, the latency of the original atomic broadcast increases with the system size. This increase is linear when the throughput is low as shown in Figures 9.7(a) and 9.7(c): the main influence on the latency of atomic broadcast is the $O(n)$ acknowledgments that need to be received and processed (by the coordinator in the case of *CT*, by all processes in the case of *MR*) before deciding. The increase is quadratic when the throughput is higher as shown in Figures 9.7(b) and 9.7(d): the $O(n^2)$ messages needed by reliable broadcast (and that have to be sent and processed by the different processes) start to influence the latency of atomic broadcast.

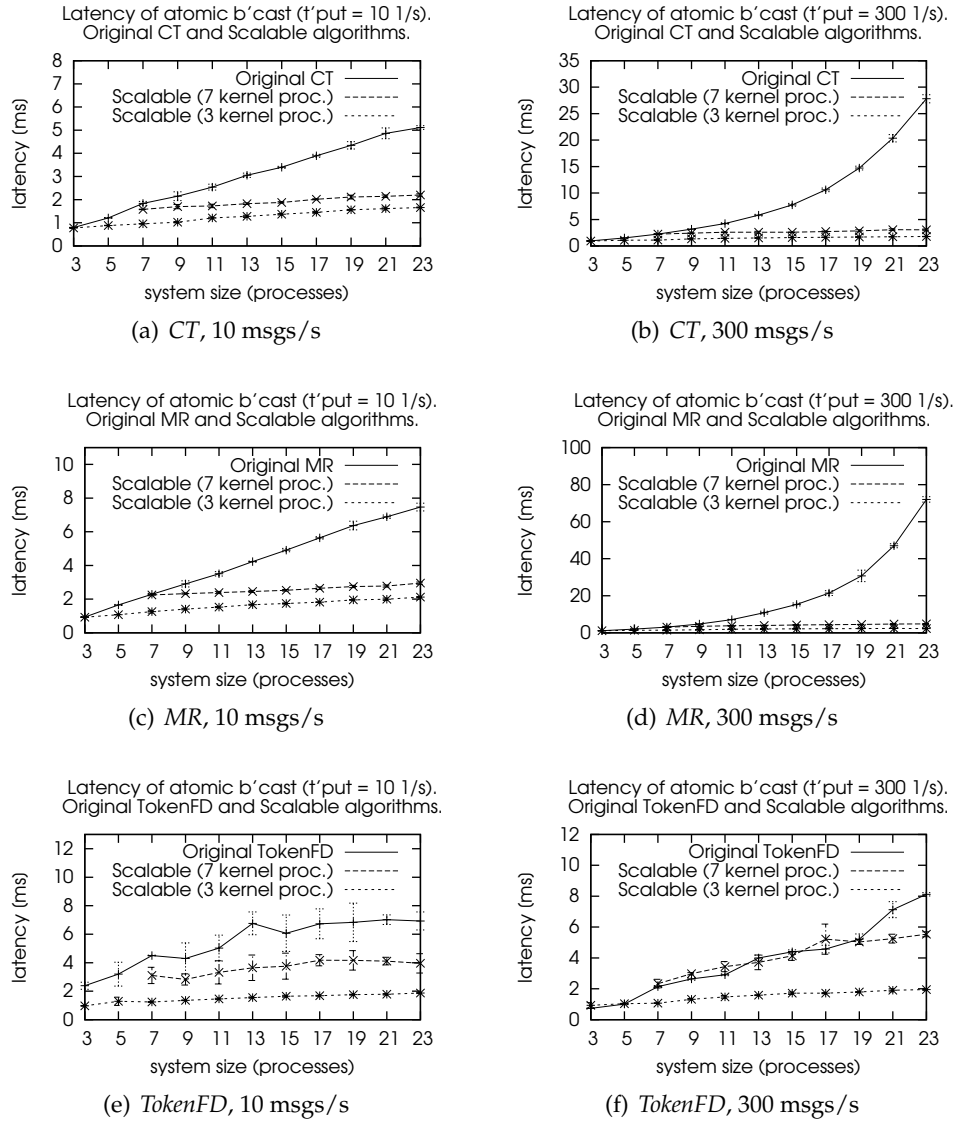


Figure 9.7: Latency vs. system size of the original and scalable atomic broadcast algorithms, for two different throughputs on the local area network.

The second trend in Figure 9.7 is the slow increase of the latency of the scalable atomic broadcast algorithm as the system size increases. The figure clearly shows that the latency of the scalable algorithm is mainly influenced by the kernel size; the additional processes only moderately affect its performance.

These results strongly confirm the intuition behind the scalable atomic broadcast algorithm: if only a fixed subset of the entire system executes the agreement algorithm, then the local area network system can grow in the number of processes with a limited impact on the performance of atomic broadcast.

9.3.3.B Wide Area Network: Grid'5000

In the previous paragraphs, we showed that the scalable atomic broadcast algorithm has interesting performance characteristics as the size of a local area network system increases.

The second phase of the experimental validation of the algorithm consists in measuring its performance on a wide area network. The intent of this second validation is to examine how the performance of the scalable atomic broadcast algorithm is affected if the kernel processes need to communicate through channels with higher latencies. As explained in Section 9.3.2.B, the processes are distributed on 7 different sites and consequently, we set up the system so that kernel processes are always on different sites (for both kernel sizes $k = 3$ and $k = 7$). The $n - k$ other processes are evenly distributed among the 7 sites.

The original algorithm: Figure 9.8 shows the latency versus the throughput of the three atomic broadcast algorithms running on all processes, on the Grid'5000 wide area network. The figure shows results for system sizes ranging from three to 23 processes, supporting from 1 to 11 failures (*CT* and *MR*) or 1 to 4 failures (*TokenFD*).

For a given system size and a given algorithm, the latency of atomic broadcast remains relatively stable until a threshold is reached. Above that threshold, the latency quickly increases. This is especially the case for the smaller systems ($n = 3$ to $n = 15$). Again, in a wide area network, the atomic broadcast algorithms do not scale well as the number of processes in the system and the throughput increase.

In the wide area network however, the additional communication steps (which are costly) needed by the *TokenFD* algorithm as the system size increases strongly affect the performance of the algorithm. Figure 9.8(c) shows that if a single failure is supported ($n = 3$), then the maximum throughput of the algorithm is above 3250 msgs/s. This threshold drops to approximately 2000 msgs/s if two failures are supported ($n = 7$ and 11) and does not exceed 500 msgs/s when four failures are supported ($n = 23$).

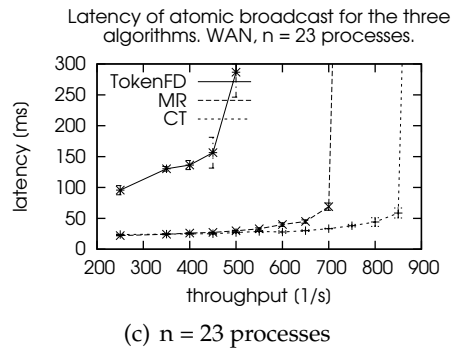
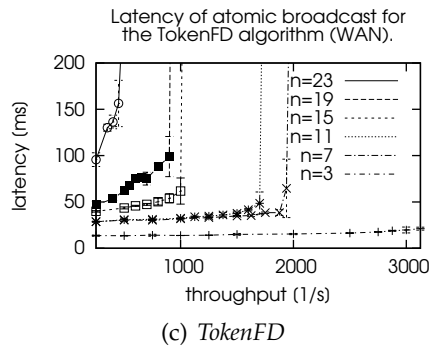
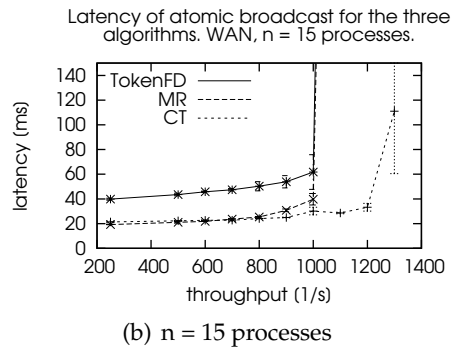
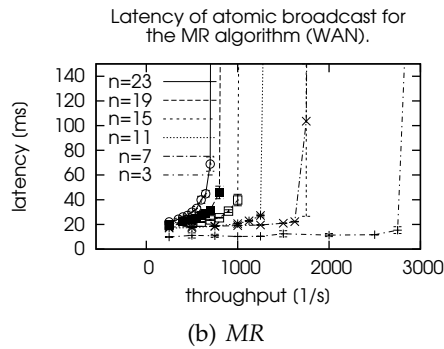
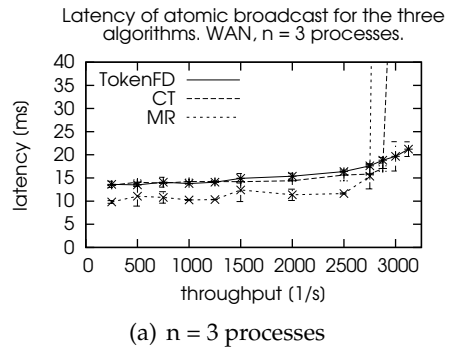
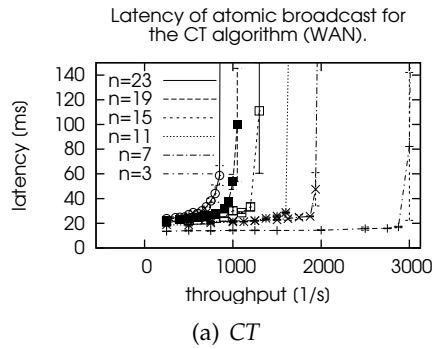


Figure 9.8: Latency vs. throughput of the Chandra-Toueg (with CT and MR consensus) and TokenFD atomic broadcast algorithms in the Grid'5000 wide area network setup.

Figure 9.9: Latency vs. throughput of the three atomic broadcast algorithms for system sizes of 3, 15 and 23 processes in the Grid'5000 wide area network.

Each additional supported failure requires an additional communication step to *deliver* a message. On average, such a communication step takes around 12 ms with the token circulation path that we consider.

Finally, Figure 9.9 compares the impact of the system size on the three algorithms. When $n = 3$ (Figure 9.9(a)), the three algorithms have similar latencies (with a slight advantage for *MR*, that needs less communication steps) and *TokenFD* reaches higher throughputs than both other algorithms. The increase of the system size (Figures 9.9(b) and 9.9(c)) has the strongest negative effect on the latency of the *TokenFD* algorithm (due to the additional expensive communication steps). The *MR* algorithm still has a slightly lower latency than *CT* for low throughputs but cannot sustain as high throughputs as *CT* (since it sends more messages than *CT*).

The scalable algorithm: Figures 9.10 and 9.11 show the latency as a function of the throughput of the scalable atomic broadcast algorithm in the Grid'5000 wide area network setting. Two kernel sizes, $k = 3$ and $k = 7$, supporting respectively 1 and 3 (*CT*, *MR*) or 2 (*TokenFD*) failures, are considered. This time, the dependency between latency and throughput is similar for all system sizes: the latency is almost constant for all throughputs, both for a kernel size of 3 and 7. The latency of the scalable atomic broadcast algorithm is once again (as in the case of the local area network system) strongly correlated to the latency of the underlying atomic broadcast on the kernel processes and is hardly influenced by the total number of processes in the system (as long as the highest throughputs are not considered).

Finally, Figure 9.12 shows the latency of atomic broadcast as a function of the number of processes in the system, for two fixed throughputs and all three atomic broadcast algorithms. For the *CT* and *MR* algorithms, in the case of a small throughput (Figures 9.12(a) and 9.12(c)), the original and scalable atomic broadcast algorithms achieve similar results. The reason is the following: the consensus coordinator in *CT* and all processes in *MR* need to wait for $\lceil \frac{n+1}{2} \rceil$ acknowledgments in each execution to decide. Since all 7 sites have different round-trip times pair-wise and that the processes are evenly distributed among the sites, the processes receive the last needed ack from the site with the median round-trip time. Thus, whether 7 or 21 processes (3 processes on each of the 7 locations) participate in the algorithm, the process needs to wait a similar amount of time before having collected enough acknowledgments to decide (the processing time of the acknowledgments is negligible compared to the transmission times on the network). It is interesting to notice that for the scalable algorithm using *CT* with a kernel of size 3, the latency is slightly smaller than if a kernel of 7 processes is used. This is simply due to the fact that the median round-trip time among the 3 kernel processes (on Orsay, Bordeaux and Nancy)

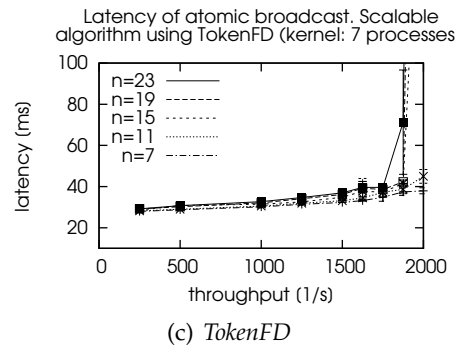
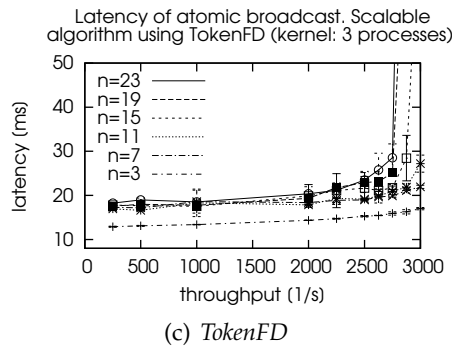
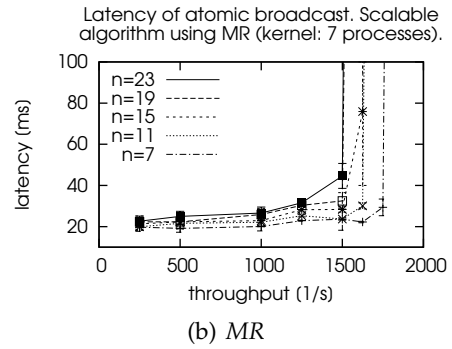
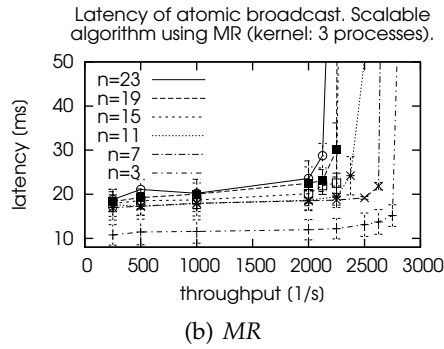
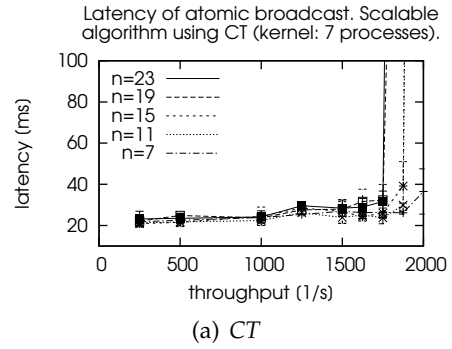
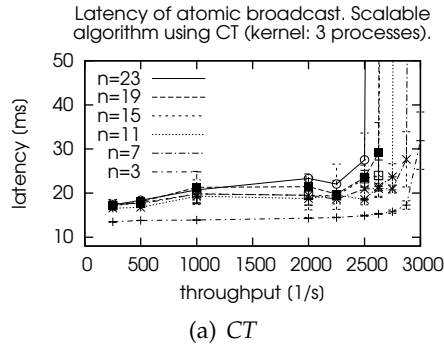


Figure 9.10: Latency vs. throughput of the scalable atomic broadcast algorithm with a kernel of 3 processes in the Grid'5000 wide area network setup.

Figure 9.11: Latency vs. throughput of the scalable atomic broadcast algorithm with a kernel of 7 processes in the Grid'5000 wide area network setup.

9.3. Performance evaluation

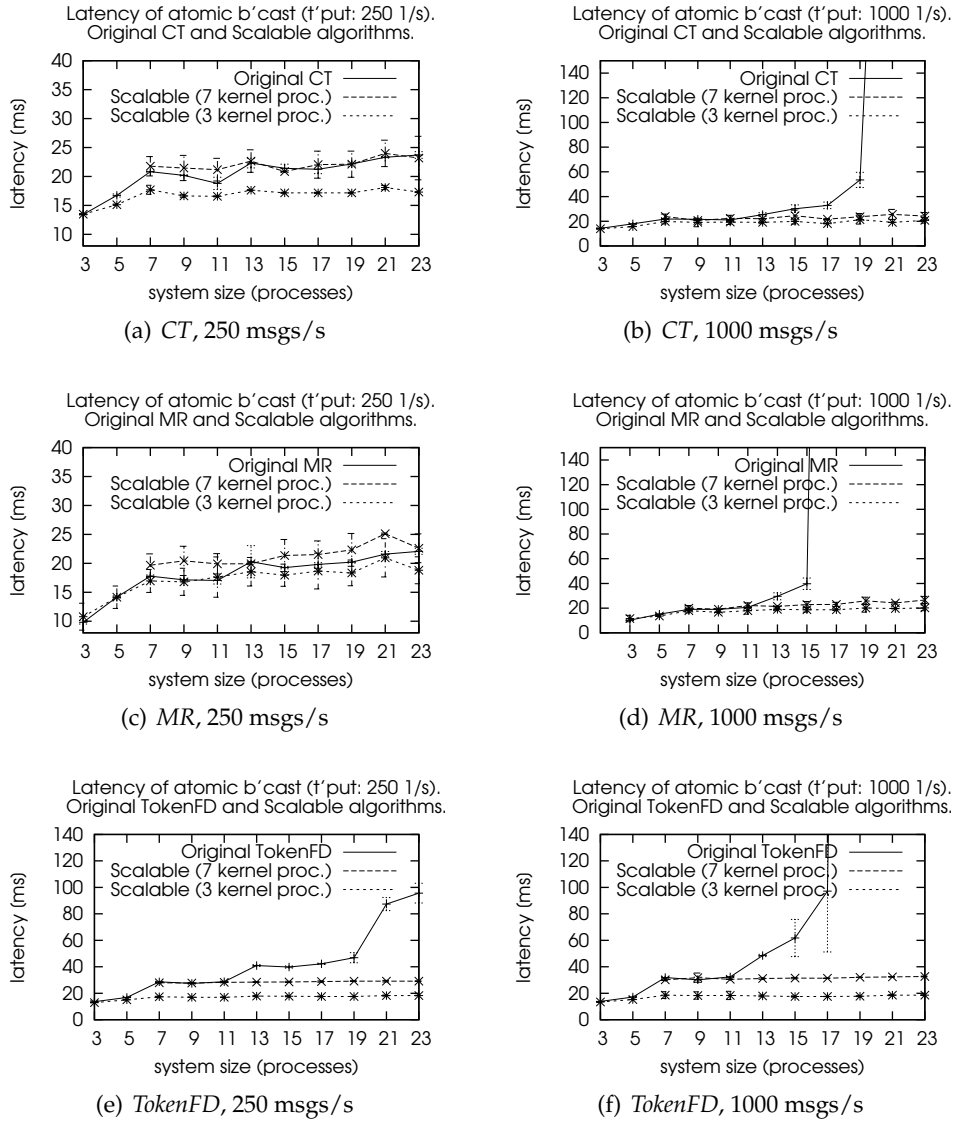


Figure 9.12: Latency vs. system size of the original and scalable atomic broadcast algorithms, for two different throughputs on the Grid'5000 Wide Area Network

is 5.7 ms, whereas the median among all 7 sites is 8.3 ms (the Bordeaux site). In the case of the *TokenFD* algorithm (Figure 9.12(e)), the number of supported failures determines the performance of the algorithm. There are four sets of values for n inside which the performance is almost identical: $n \in \{3, 5\}$ (a single supported failure), $n \in \{7, 9, 11\}$ (two possible failures), $n \in \{13, 15, 17, 19\}$ (three failures) and $n \in \{21, 23\}$ (four possible failures).

In the case of a higher throughput of 1000 msgs/s, the scalable atomic broadcast algorithm outperforms the original algorithms. In the case of *CT* and *MR* (Figures 9.12(b) and 9.12(d)), the original algorithm cannot sustain a throughput of 1000 msgs/s when the system size exceeds 19 and 15 processes respectively. The latency of the scalable algorithm on the other hand remains stable for all system sizes, since the scalable algorithm avoids the bottleneck of the $O(n^2)$ messages that need to be transmitted and processed. In the case of *TokenFD*, the original atomic broadcast algorithm cannot sustain a throughput of 1000 msgs/s if the size of the system is greater than 17 processes. The latency of the scalable algorithm using *TokenFD* on the kernel processes is however stable for all the system sizes that were considered. As was the case in the local area network, this is again due to the fact that in the scalable atomic broadcast algorithm, the number of communication steps needed to *adeliver* a message is determined by the size of the kernel, which is independent of the system size.

Compared to the original algorithm, the scalable atomic broadcast algorithm allows higher throughputs to be reached in large systems, as was the case in the local area network setting. These results again confirm the intuition behind the scalable atomic broadcast algorithm: if the execution of the agreement algorithm is limited to a subset of all processes, the atomic broadcast algorithm scales well as the total number of processes in the system increases, even if the kernel processes communicate through links with large round-trip times.

This confirms that to achieve scalable performance characteristics, only a subset of all processes should execute the actual agreement algorithm.

9.4 Discussion

Many atomic broadcast algorithm based on failure detectors have been published during the years. The performance of these algorithms has often been evaluated in small systems and the algorithms were criticized for not scaling well. A solution to the scaling problem is to let a subset of all processes execute the actual agreement algorithm and have them send updates to all other processes. This solution has two potential benefits: it reduces the impact of the system size on both the message complexity and the time complexity.

In this chapter, we presented a formal description of this scalable atomic

broadcast algorithm, as well as its proof of correctness. We then evaluated its performance in two settings: a local area network and a wide area network of 7 interconnected clusters.

The evaluation in a local area network clearly shows that the scalable atomic broadcast algorithm is only moderately affected by the system size, whereas the impact of the system size on the latency of the original atomic broadcast algorithm is much higher. The performance of the scalable atomic broadcast algorithm is mainly influenced by the size of the kernel of processes that participated in the agreement protocol.

In the wide area network, the main factor influencing the latency of atomic broadcast is the round-trip times between the sites. The impact of the scalable atomic broadcast algorithm compared to the underlying atomic broadcast algorithm is thus the highest for the *TokenFD* algorithm which needs $O(\sqrt{n})$ communication steps to solve atomic broadcast (whereas *CT* and *MR* need a constant number of steps). For high throughputs, the performance of the scalable atomic broadcast algorithm is much less affected than the performance of all three underlying algorithms as the size of the system increases.

The intuition behind the scalable atomic broadcast algorithm is verified by the performance measurements: if the size of the system increases, the efficient approach is to let a subset of the processes execute the actual agreement algorithm. This in turn validates limiting performance measurements to relatively small systems: the main factor behind the performance of atomic broadcast in a large system is the performance of the agreement algorithm on the kernel. The kernel is a relatively small subset of the processes in the system and can be tailored to achieve the fault-tolerance required by the application that uses the group communication middleware.

Conclusion

10.1 Research assessment

In Part I, we compared quorum systems and group communication, two techniques that help in addressing the problem of replication and pointed out in which case either technique is best adapted. We then presented a new atomic broadcast algorithm which uses a token to order messages and a failure detector to tolerate process failures. We also discussed how to adapt Chandra and Toueg's atomic broadcast algorithm to run consensus on message identifiers instead of messages. In Part II, we started by describing the implementation of quasi-reliable channels using a session layer protocol on top of TCP. We then evaluated the performance of several atomic broadcast algorithms in various settings, including local and wide area networks. Each one of these contributions are now assessed in further detail.

10.1.1 Atomic broadcast

Replication: Understanding the advantage of atomic broadcast over quorum systems Atomic broadcast and quorum systems are two techniques that hide some of the difficulties of replication, where clients interact with a replicated server. We showed that the two techniques are useful in different situations, depending on the desired level of isolation on read and write operations: no isolation, read-write isolation (where reading a variable and assigning it a new value are executed atomically) and general isolation (where any sequence of read and write operations are executed in isolation).

When no isolation is required, quorum systems are well adapted. Read-write isolation on the other hand, is easier to implement with atomic broadcast than with quorums (more precisely, atomic broadcast requires a $\diamond S$ failure detector, whereas a quorum system requires the stronger \mathcal{P} failure

detector). Finally, the atomic broadcast solution can also be extended to handle general isolation if (1) the function that updates the data on the servers can be defined statically and (2) the identity of the servers to which the update function must be sent is known statically. In other cases, the atomic broadcast solution may not be applicable, in which case quorum systems should be used.

A token based atomic broadcast algorithm We presented an atomic broadcast algorithm, noted *TokenFD*, that orders messages using a token and tolerates failures with a new unreliable failure detector \mathcal{R} , specifically targeted at (logical) ring network topologies. The algorithm was simulated and compared to two other implementations of atomic broadcast using failure detectors: *CT* and *MR*. The results showed that the performance of *TokenFD* was superior to both other algorithms, especially for high system loads (where token based algorithms usually perform well) or when frequent wrong failure suspicions occurred (a situation well-adapted to algorithms using failure detectors).

We also discussed several optimizations to the *TokenFD* algorithm, that reduce the number of tokens sent or bound the amount of memory that is needed by the algorithm. Finally, the algorithm was also presented in the *Heard-Of* model.

Solving atomic broadcast with indirect consensus Atomic broadcast can be reduced to consensus on messages, as shown by Chandra and Toueg. However, in this reduction, the performance of the consensus algorithm is affected by the (potentially) large messages. As a consequence, a more efficient reduction is for atomic broadcast to use consensus on message identifiers.

We showed that the reduction of atomic broadcast to consensus on message identifiers can lead to faulty executions and presented *indirect* consensus that addresses this issue. Adapting a consensus algorithm to indirect consensus is not always trivial: the resilience of the Mostéfaoui-Raynal consensus algorithm is reduced when transformed into an indirect consensus algorithm. On the other hand, the consensus algorithm by Chandra and Toueg is easily adapted.

Finally, we showed that the performance of the reduction of atomic broadcast to indirect consensus is better than two other correct approaches (the reduction of atomic broadcast to consensus on messages, or to consensus on message identifiers, but using uniform reliable broadcast for the message diffusion).

10.1.2 Experimental performance evaluations

Robust TCP: implementing quasi-reliable channels Quasi-reliable communication channels are commonly used by group communication algorithms. The widespread TCP protocol addresses some of the issues that are inherent to the implementation of quasi-reliable channels, such as avoiding duplication or spontaneous creation of messages. However, TCP does not address link failures that last more than a couple of minutes.

We thus presented a session-layer protocol above TCP (called *Robust TCP*) that implements quasi-reliable channels by handling link failures. The design of the protocol was presented and a prototype was implemented in Java. Performance measurements in a local area network show that the Robust TCP protocol introduces less than 10% overhead over regular TCP connections.

Comparing atomic broadcast algorithms in a local area network

Experimental evaluation is important when assessing the performance of a distributed algorithms. Consequently, the second part of this work focused on the performance evaluation of several group communication algorithms in various real systems.

The first setting considered in this work was a local area network. The comparison showed that when no failures nor suspicions occur in a system supporting one or two failures, *TokenFD* achieves higher throughputs and lower latencies than *CT* and *MR*. Furthermore, when wrong suspicions occur (but no processes fail) the latency of the *TokenFD* algorithm is also lower than *CT* and *MR*. These experimental results confirmed the properties of *TokenFD* that were already observed in simulation: the token allows high throughputs, whereas wrong suspicions are supported well with the failure detector.

Finally, *TokenFD* was compared to a group membership and token based algorithm, namely *MovingSeq*. In this second comparison, *TokenFD* reached lower throughputs than *MovingSeq* in a system without failures nor suspicions. However, we also explained that whenever a suspicion occurs, the group membership protocol needs to execute several costly operations and the performance of *MovingSeq* is thus strongly affected by frequent wrong suspicions.

Evaluating the performance of atomic broadcast algorithms in high latency networks

The second setting in which we evaluated atomic broadcast algorithms was wide area networks systems with three processes. We started by modeling the *TokenFD*, *CT* and *MR* algorithms in a wide area network with three processes distributed on two or three different locations.

The algorithms were then experimentally evaluated. This evaluation showed that our simple model accurately predicts the performance of the

three algorithms for moderate system loads. Furthermore, the main characteristic that influenced the performance of the algorithms was the number of communication steps needed to *adeli*ver messages. This was not surprising for the wide area network with large round trip times (300 *ms*) between locations, but was also true for the network with the lowest round trip times (4 *ms*).

Finally, we also showed that some parameters, such as the choice of the coordinator site in the consensus algorithms, have a large impact on performance. In order to achieve high throughputs, it is advisable to choose parameters that allow the process on the distant location to actively participate in the ordering of messages.

On the scalability of atomic broadcast algorithms Finally, we evaluated the impact of the system size on the performance of the three failure detector based atomic broadcast algorithms *TokenFD*, *CT* and *MR*. Local area and wide area networks were considered.

The evaluation showed that all three algorithms were affected by the system size. When limiting the actual ordering algorithm to a (fixed) subset of the system (the *kernel*) we showed that the performance of atomic broadcast was hardly affected by the addition of (non-kernel) processes to the system. The kernel can thus be tailored to suit the needs of the application, and additional processes (acting as caches of the distributed state) can be added without affecting the performance of the system.

10.2 Open questions and future research directions

Replication: Understanding the advantage of atomic broadcast over quorum systems The comparison between atomic broadcast and quorum systems showed in which situations each technique is most adapted. This comparison could be extended by experimentally evaluating the performance of both techniques.

These techniques both use different approaches to updating data: in quorum systems, the servers and clients exchange data. The clients modify the data and then inform the servers of the new state of the data. In contrast, in an atomic broadcast system, the clients instead invoke procedures on the servers to update data. The modification of the data is done directly by the server and no data needs to be exchanged.

A token based atomic broadcast algorithm Future work on the token and failure detector based atomic broadcast algorithm includes further optimizing the token processing. Currently, whenever a token is received,

relatively costly operations need to be performed on the data structures contained in the token. Since the rate of *adelivers* (when the load on the system is high) is inversely proportional to the time to receive, handle and send the token, optimizing the token processing directly affects the performance of the algorithm in experimental evaluations.

Furthermore, the simulation model that was used to evaluate the *TokenFD* algorithm aims at modeling network and CPU contention as simply as possible. This model could be extended to take factors such as message size, processing complexity of messages and inter-process round-trip times into account. These factors would add some complexity to the simulation model, but could yield more realistic simulation results.

Solving atomic broadcast with indirect consensus We transformed two consensus algorithms into indirect consensus algorithms. Transforming the *CT* consensus algorithm was easy, whereas the transformation of *MR* affected the resilience of the algorithm (the indirect consensus algorithm supported at most $f < \frac{n}{3}$ process failures whereas the original algorithm supported $f < \frac{n}{2}$). It would be interesting to examine how other consensus algorithms are affected when transformed into indirect consensus algorithms. Furthermore, this could possibly lead to a more general classification criterion indicating which consensus algorithms are affected by a transformation into indirect consensus.

Robust TCP connections: implementing quasi-reliable channels The Robust TCP sockets that we presented reimplement the interface of the standard Java TCP. As this interface evolves (with the different versions of Java), the Robust TCP implementation must be kept up to date. Furthermore, a tighter integration with the Java sockets is also planned, in order to minimize the source code modifications needed to replace standard sockets by Robust sockets in an existing application.

Finally, as mentioned in Chapter 6, an implementation of the Robust sockets in another language is also possible. It would be interesting, for example, to provide Robust TCP sockets implemented in C (which is used for a number of group communication implementations).

Comparing atomic broadcast algorithms in a local area network We compared different atomic broadcast algorithms in several settings. Future research contributions include extending these experimental evaluations. Different workloads can for example be considered: in the experiments that we presented, the processes sent messages following a Poisson stochastic process. Other sending patterns, exposing bursty characteristics or generated from observed usage patterns, could be included to evaluate the atomic broadcast algorithms in possibly more realistic conditions.

Furthermore, the evaluations included in this work mainly focused on failure detector based algorithms (although one group membership based algorithm was also examined). In the future, the performance evaluation of atomic broadcast could be extended to other algorithms, using group membership for example. Algorithms with different resilience levels (at most $f < \frac{n}{3}$ possible failure for example) are also interesting candidates.

Finally, additional faultloads could add value to the comparison between algorithms. The difference, for example, between failure detector and group membership based algorithms could be evaluated in systems where frequent wrong suspicions occur. Additionally, the short and long term impacts of process crashes on the performance of the algorithms could also be assessed.

Evaluating the performance of atomic broadcast algorithms in high latency networks The evaluation of atomic broadcast in wide area networks was conducted in systems with three processes. This in turn allowed a preliminary analytical evaluation of three algorithms. In future work, the size of the system could be extended. The analytical evaluation of the algorithms is then more complex (especially if more than three different locations are considered) and would probably be replaced by a numerical performance evaluation. The simulation model of the Neko framework could be extended to address this issue.

Additionally, we showed that the performance of the algorithms was affected by the bandwidth of the wide area network links. Consequently, the simple model that we presented could be extended to take the bandwidth of the links into account and thus reduce some of the discrepancies between the currently modeled and experimental results.

On the scalability of atomic broadcast algorithms We evaluated the performance of atomic broadcast algorithms in systems with a large number of processes. Future research directions include an evaluation of the scalable atomic broadcast algorithm in the context of an actively replicated database. Indeed, one of the advantages of the scalable algorithm is that the non-kernel processes act as a cache of the state of the system. These non-kernel processes can thus relieve the load on the kernel processes by handling queries that do not need to be *abroadcast* (e.g. a read-only query that accesses the local state of one of the replicas).

Furthermore, the set of kernel processes needs not necessarily be fixed. A second research direction is thus to examine how to dynamically adapt the kernel set as a response to process failures, for example. The goal is to remove failed processes from the kernel in order to maintain the fault tolerance degree that the kernel was tailored for.

Bibliography

- [ABEK⁺01] L. Alvisi, T. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *Proc. 20th Annual Joint Conf. of the IEEE Computer and Communications Societies (Infocom)*, Anchorage, AK, USA, April 2001.
- [ADGFT00] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC'2000)*, October 2000.
- [ADGS03] T. Anker, D. Dolev, G. Greenman, and I. Shnayderman. Evaluating total order algorithms in WAN. In *Proc. International Workshop on Large-Scale Group Communication*, Florence, Italy, October 2003.
- [AM98] L. Alvisi and K. Marzullo. Waft: Support for fault-tolerance in wide-area object oriented systems. In *Proceedings of the 2nd Information Survivability Workshop – ISW '98*, pages 5–10, Los Alamitos, CA, USA, October 1998. IEEE Computer Society Press.
- [AMMS⁺95] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, November 1995.
- [Avi85] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.
- [BCBT96] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Int. Workshop on Distributed Algorithms (WDAG'96)*, LNCS 1151, pages 105–122, Bologna, October 1996.
- [Ben83] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Sec-*

- ond ACM Symposium on Principles of Distributed Computing (PODC-2), pages 27–30, Montreal, Canada, August 1983.
- [BK02] O. Bakr and I. Keidar. Evaluating the running time of a communication round over the internet. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 243–252, Monterey, California, USA, 2002. ACM Press.
- [BL91] A. Burns and A. M. Lister. A framework for building dependable systems. *The Computer Journal*, 34(2):173–181, 1991.
- [BS97] K. Brown and S. Singh. M-TCP: TCP for mobile cellular networks. *ACM Computer Communication Review*, 27(5), October 1997.
- [BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, August 1991.
- [BT93] O. Babaoğlu and S. Toueg. Non-blocking atomic commitment. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 6, pages 147–168. Addison-Wesley, second edition, 1993.
- [Bur06] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, Seattle, WA, USA, November 2006.
- [CB03] B. Charron-Bost. Comparing the atomic commitment and consensus problems. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing, Research and Position Papers*, volume 2584 of *Lecture Notes in Computer Science*, pages 29–34. Springer, 2003.
- [CBDS02] B. Charron-Bost, X. Défago, and A. Schiper. Broadcasting Messages in Fault-Tolerant Distributed Systems: The benefit of handling input-triggered and output-triggered suspicions differently. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 244–249, Osaka, Japan, October 2002.
- [CBS06] B. Charron-Bost and A. Schiper. The Heard-Of Model: Unifying all Benign Failures. Technical Report LSR-Report-2006-004, EPFL, Lausanne, Switzerland, 2006.

- [CC98] S. Chandra and P. M. Chen. How fail-stop are faulty programs? In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, pages 240–249, Munich, Germany, June 1998. IEEE Computer Society.
- [CCD⁺05] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000: a large scale, re-configurable, controlable and monitorable Grid platform. In *Grid'2005 Workshop*, Seattle, USA, November 13-14 2005. IEEE/ACM.
- [CD89] B. Chor and C. Dwork. Randomization in Byzantine agreement. In S. Micali, editor, *Advances in Computing Research, Randomness in Computation*, volume 5, pages 443–497. JAI Press, 1989.
- [CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. on Parallel & Distributed Systems*, 10(6):642–657, June 1999.
- [CFM⁺97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java Just in Time. *IEEE Micro*, 17(3):36–43, 1997.
- [CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of ACM*, 43(4):685–722, July 1996.
- [CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *Computing Surveys*, 4(33):1–43, December 2001.
- [CL99] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.
- [CM84] J. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. 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.

Bibliography

- [CMA97] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distributed System Engineering Journal*, 4(2):109–128, June 1997.
- [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 ACM*, 43(2):225–267, 1996.
- [CTA02] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(2):561–580, May 2002.
- [CUBS02] A. Coccoli, P. Urbán, A. Bondavalli, and A. Schiper. Performance analysis of a consensus algorithm combining Stochastic Activity Networks and measurements. In *Proc. Int’l Conference on Dependable Systems and Networks (DSN 2002)*, pages 551–560, Washington, DC, USA, June 2002.
- [DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of ACM*, 34(1):77–97, January 1987.
- [DLS88] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.
- [DSU04] X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(2):372–421, December 2004.
- [ES03] R. Ekwall and A. Schiper. Revisiting token-based atomic broadcast algorithms. Technical Report IC/2003/39, École Polytechnique Fédérale de Lausanne, Switzerland, February 2003.
- [ESU04] R. Ekwall, A. Schiper, and P. Urbán. Token-based atomic broadcast using unreliable failure detectors. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS)*, Florianópolis, Brazil, October 2004.

- [Fel98] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.
- [Fet00] C. Fetzer. Enforcing synchronous system properties on top of timed systems. In *Proc. 7th IEEE Pacific Rim Symposium on Dependable Computing (PRDC-7)*, pages 185–192, Los Angeles, California, USA, December 2000. IEEE Computer Society.
- [Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical Report 273, Department of Computer Science, Yale University, New Haven, Conn., USA, June 1983.
- [FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, April 1985.
- [FR03] F. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.
- [FRT99] E. Fromentin, M. Raynal, and F. Tronel. On classes of problems in asynchronous distributed systems with process crashes. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS-19)*, pages 470–477, Austin, TX, USA, June 1999. IEEE Computer Society Press.
- [Gif79] D. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–159, December 1979.
- [GLPQ06] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. High Throughput Total Order Broadcast for Cluster Environments. In *IEEE International Conference on Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, USA, June 2006.
- [GS01] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science (Elsevier)*, 254:297–316, 2001.
- [Hen99] J. Hennessy. The future of systems research. *IEEE Computer*, 32(8):27–33, 1999.
- [Her88] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM Symposium on Principles of Distributed Computing (PODC)*, pages 276–290, August 1988.

Bibliography

- [Her91] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.
- [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.
- [IBM00] IBM Corporation. *SockPerf: A Peer-to-Peer Socket Benchmark Used for Comparing and Measuring Java Socket Performance*, 2000. <http://www.alphaWorks.ibm.com/aw.nsf/techmain/sockperf>.
- [ISO96] ISO. *Information technology – Open Systems Interconnection – Connection-oriented Session protocol: Protocol specification*. ISO/IEC 8327-1. International Organization for Standards, 1996.
- [Jah94] F. Jahanian. Fault-tolerance in embedded real-time systems. In *Papers of the workshop on Hardware and software architectures for fault tolerance: experiences and perspectives*, pages 237–249, Le Mont Saint Michel, France, 1994. Springer-Verlag.
- [KG94] H. Kopetz and G. Grünsteidl. TTP-a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, 1994.
- [KMA02] K. Kanoun, H. Madeira, and J. Arlat. A Framework for Dependability Benchmarking. In *Workshop on Dependability Benchmarking (jointly organized with DSN-2002)*, pages F7–F8, Bethesda, Maryland, USA, June 2002.
- [LAF99] M. Larrea, S. Arevalo, and A. Fernandez. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *International Symposium on Distributed Computing*, pages 34–48, 1999.
- [Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 1978.
- [Lam86] L. Lamport. On interprocess communications, i, ii. *Distributed Computing*, 1(2):77–101, October 1986.
- [Lam98] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, 1998.
- [Lan97] G. L. Lann. An analysis of the Ariane 5 flight 501 failure—a system engineering perspective. In *Proceedings of the International*

-
- Conference and Workshop on Engineering of Computer-Based Systems (ECBS '97)*, pages 339–246, Monterey, CA, USA, March 1997. IEEE Computer Society.
- [Lar03] M. Larrea. On the weakest failure detector for hard agreement problems. *Journal of Systems Architecture*, 49(7):345–353, October 2003.
- [LKPMJP05] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Consistent data replication: Is it feasible in WANs?. In *Proc. 11th International Euro-Par Conference*, pages 633–643, Lisbon, Portugal, September 2005.
- [Lyn89] N. Lynch. A hundred impossibility proofs for distributed computing. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC-8)*, pages 1–28, Edmonton, Alberta, Canada, August 1989. ACM Press.
- [Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [MBK⁺05] S. Mena, C. Basile, Z. Kalbarczyk, A. Schiper, and R. K. Iyer. Assessing the crash-failure assumption of group communication protocols. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 107–116, Chicago, Illinois, USA, November 2005. IEEE Computer Society.
- [MFVP05] N. Mittal, F. C. Freiling, S. Venkatesan, and L. D. Penso. Efficient reduction for wait-free termination detection in a crash-prone distributed system. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, pages 93–107, Cracow, Poland, September 2005.
- [MK00] S. Mishra and S. M. Kuntur. Newsmonger: A technique to improve the performance of atomic broadcast protocols. *Journal of Systems and Software*, 55(2):167–183, December 2000.
- [MR98] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, 1998.
- [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. Springer-Verlag.

Bibliography

- [MS01] N. F. Maxemchuk and D. H. Shur. An Internet multicast system for the stock market. *ACM Trans. on Computer Systems*, 19(3):384–412, August 2001.
- [MSW03] S. Mena, A. Schiper, and P. Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of Int'l Middleware Conference*, pages 414–432. Springer, June 2003.
- [Nag84] J. Nagle. RFC 896: Congestion control in IP/TCP internet networks, January 1984. Status: UNKNOWN.
- [NF97] N. Neves and W. Fuchs. Fault detection using hints from the socket layer. In *Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS)*, pages 64–71, Durham, North Carolina, USA, October 1997. IEEE.
- [PSUC02a] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proc. 4th European Dependable Computing Conference (EDCC-4)*, Toulouse, France, October 2002. To appear.
- [PSUC02b] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Weak ordering oracles for failure detection-free systems. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN), supplementary volume*, pages B–32–33, June 2002. Fast abstract.
- [Ran75] B. Randell. System structure for fault tolerance. *IEEE Transactions on Software Engineering*, 1:220–232, 1975.
- [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 *Proceedings of the 7th IEEE International Conference on Computer Communications and Networks (IC3N'98)*, pages 840–847, Lafayette, Louisiana, USA, October 1998.
- [RM98] K. Ratnam and I. Matta. WTCP: An efficient transmission control protocol for networks with wireless links. *Proc. Third IEEE Symposium on Computers and Communications (ISCC '98)*, Athens, Greece, June 1998.
- [Sch93a] F. B. Schneider. Replication Management using the State-Machine Approach. In S. Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.

- [Sch93b] F. B. Schneider. What good are models and what models are good? In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 2, pages 17–26. Addison-Wesley, second edition, 1993.
- [Sch03] A. Schiper. Practical Impact of Group Communication Theory. In *Future Directions in Distributed Computing*, pages 1–10. Springer, LNCS 2584, 2003.
- [Soc99] Shortcomings of SocketImplFactory. Bug report on Sun’s Java Developer Connection site, 1999. <http://developer.java.sun.com/developer/bugParade/bugs/4245730.html>.
- [SPMO02] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic Total Order in Wide Area Networks. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 190–199, Osaka, Japan, October 2002.
- [SS83] R. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. on Computer Systems*, 1(3):222–238, August 1983.
- [Ste94] R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, January 1994.
- [SXM⁺00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. *IETF*, April 2000.
- [Tho79] R. Thomas. A majority consensus approach to concurrency control for multiple copies databases. *ACM Trans. on Database Systems*, 4(2):180–209, 1979.
- [TLS05] P. Traverse, I. Lacaze, and J. Souyris. A process toward total dependability - airbus fly-by-wire paradigm. In M. D. Cin, M. Kaâniche, and A. Pataricza, editors, *Dependable Computing – EDCC-5*, volume 3463 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag Inc., 2005.
- [UDS00] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proc. 9th IEEE Int’l Conf. on Computer Communications and Networks (IC3N 2000)*, pages 582–589, October 2000.
- [UDS02] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms.

- Journal of Information Science and Engineering*, 18(6):981–997, November 2002.
- [UHSK04] P. Urbán, N. Hayashibara, A. Schiper, and T. Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 4–17, Florianópolis, Brazil, October 2004.
- [Urb03] P. Urbán. *Evaluating the Performance of Distributed Agreement Algorithms: Tools, Methodology and Case Studies*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, August 2003. Number 2824.
- [USS03] P. Urbán, I. Shnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *Proc. of the Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 645–654, June 2003.
- [VR02] P. Vicente and L. Rodrigues. An Indulgent Total Order Algorithm with Optimistic Delivery. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 92–101, Osaka, Japan, October 2002.
- [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.
- [WV02] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.
- [ZD95] Y. Zhang and S. Dao. A persistent connection model for mobile and distributed systems. In *Proc. 4th Int'l Conf. on Computer Communications and Networks (ICCCN)*, Las Vegas, NV, USA, September 1995.

Agreement algorithms

The following section describes the consensus, reliable broadcast and atomic broadcast algorithms that are considered in this thesis. We present the pseudo code of the algorithms, as well as a short analysis of their time and message complexities. Reliable broadcast and consensus algorithms are introduced first, since they serve as building blocks for one of the atomic broadcast algorithms.

A.1 Reliable broadcast

The reliable broadcast algorithm that is considered in most of the chapters is proposed in [CT96] and is presented in Algorithm A.1.

Algorithm A.1: Reliable broadcast algorithm (code of process p)

```
1: procedure rbroadcast( $m$ )  
2:   send  $m$  to all (including  $p$ )  
  
3: when receive  $m$  for the first time {rdeliver occurs as follows}  
4:   if  $\text{sender}(m) \neq p$  then  
5:     send  $m$  to all  
6:   rdeliver( $m$ )
```

Whenever a message m is *rbroadcast*, it is sent to all processes (including the sender). Then, when a process p receives a message m for the first time, p retransmits it to everyone (if p is not the original sender) and then *rdelivers* m .

This reliable broadcast algorithm needs one communication step and sends n^2 messages on the network for each *rbroadcast*.

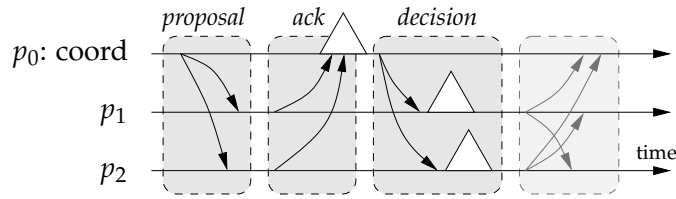


Figure A.1: Communication pattern of the Chandra-Toueg $\diamond S$ consensus algorithm in runs without failures or wrong suspicions ($n = 3$ processes).

A.2 Consensus

In this thesis, we consider two consensus algorithms that require a majority of correct processes, quasi-reliable channels, and use the $\diamond S$ unreliable failure detector [CT96] to reach a decision. The algorithms are presented respectively by Chandra and Toueg in [CT96], and Mostéfaoui and Raynal in [MR99].

A.2.1 Chandra-Toueg consensus

The Chandra-Toueg $\diamond S$ consensus algorithm [CT96] is a centralized algorithm that uses the $\diamond S$ unreliable failure detector to ensure termination of consensus. The algorithm proceeds in rounds and is presented in Algorithm A.2.

At the beginning of each round, in Phase 1, each process sends its estimate of the decision to the process acting as a coordinator in that round (line 11). In Phase 2, the coordinator waits for a majority of estimates and selects the most recent one (based on its timestamp) and sends it to all processes (lines 15 to 19). In the first round of the algorithm, the coordinator always selects its own estimate (with a timestamp equal to 0) and the algorithm is thus optimized to skip the initial estimate exchange in Phases 1 and 2 (hence the conditions on lines 10 and 14).

During Phase 3, each process either receives the coordinator's proposal, or suspects the coordinator of having crashed (line 21). In the former case, the process sets its own estimate to the coordinator's proposal, updates its timestamp and sends a positive acknowledgment (*ack*) to the coordinator (lines 22 to 25). In the latter case, a negative acknowledgment (*nack*) is sent (lines 26 and 27). In both cases, before proceeding, the non-coordinator processes wait until (1) the coordinator is suspected or (2) a message (abort or decision) is received from the coordinator (lines 28 and 29).

In [CT96], the non-coordinator processes do *not* wait before proceeding to the next round and immediately send their estimate to the next coordinator. However, in good runs (without failures or suspicions), these messages are not useful, since a decision is always taken in the first round. By intro-

Algorithm A.2: Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm (code of proc. p)

```

1: procedure propose( $v_p$ )
2:    $estimate_p \leftarrow v_p$                                 { $p$ 's estimate of the decision value}
3:    $state_p \leftarrow undecided$ 
4:    $r_p \leftarrow 0$                                        { $p$ 's current round number}
5:    $ts_p \leftarrow 0$                                        { $ts_p$  is the last round in which  $p$  updated  $estimate_p$ }
6:   while  $state_p = undecided$  do                          {rotate through coordinators until decision reached}
7:      $r_p \leftarrow r_p + 1$ 
8:      $c_p \leftarrow (r_p \bmod n) + 1$                        { $c_p$  is the current coordinator}
9:     Phase 1:                                             {all processes  $p$  send  $estimate_p$  to the current coordinator}
10:    if  $r_p > 1$  then
11:      send ( $p, r_p, estimate_p, ts_p$ ) to  $c_p$ 
12:    Phase 2:                                             {coordinator gathers  $\lceil \frac{n+1}{2} \rceil$  estimates and proposes new estimate}
13:    if  $p = c_p$  then
14:      if  $r_p > 1$  then
15:        wait until [for  $\lceil \frac{n+1}{2} \rceil$  processes  $q$  : received ( $q, r_p, estimate_q, ts_q$ ) from  $q$ ]
16:         $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p \text{ received } (q, r_p, estimate_q, ts_q) \text{ from } q\}$ 
17:         $t \leftarrow$  largest  $ts_q$  such that  $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$ 
18:         $estimate_p \leftarrow$  select one  $estimate_q$  such that  $(q, r_p, estimate_q, t) \in msgs_p[r_p]$ 
19:        send ( $p, r_p, estimate_p$ ) to all
20:      Phase 3:                                           {all processes wait for new estimate proposed by current coordinator}
21:      wait until [received ( $c_p, r_p, estimate_{c_p}$ ) from  $c_p$  or  $c_p \in \mathcal{D}_p$ ] {query failure det.  $\mathcal{D}_p$ }
22:      if [received ( $c_p, r_p, estimate_{c_p}$ ) from  $c_p$ ] then   { $p$  received  $estimate_{c_p}$  from  $c_p$ }
23:         $estimate_p \leftarrow estimate_{c_p}$ 
24:         $ts_p \leftarrow r_p$ 
25:        send ( $p, r_p, ack$ ) to  $c_p$ 
26:      else                                                 { $p$  suspects that  $c_p$  crashed}
27:        send ( $p, r_p, nack$ ) to  $c_p$ 
28:      if  $p \neq c_p$  then   {non-coord. processes wait before proceeding to the next round}
29:        wait until  $c_p \in \mathcal{D}_p$  or received ( $abort, r_p$ ) from  $c_p$  or  $state_p = decided$ 
30:    Phase 4:   {the coordinator waits for  $\lceil \frac{n+1}{2} \rceil$  replies. If all replies adopt its estimate, the coordinator rbroadcasts a decide message}
31:    if  $p = c_p$  then
32:      wait until [for  $\lceil \frac{n+1}{2} \rceil$  processes  $q$  : received ( $q, r_p, ack$ ) or for 1 process  $q$ : ( $q, r_p, nack$ )]
33:      if [for  $\lceil \frac{n+1}{2} \rceil$  processes  $q$  : received ( $q, r_p, ack$ )] then
34:        rbroadcast( $p, estimate_p, decide$ ) to all
35:      else                                                 {a nack was received}
36:        send ( $abort, r_p$ ) to all
37:  when rdeliver( $q, estimate_q, decide$ )   {if  $p$  rdelivers a decide message,  $p$  decides accordingly.}
38:  if  $state_p = undecided$  then
39:    decide( $estimate_q$ )
40:     $state_p \leftarrow decided$ 

```

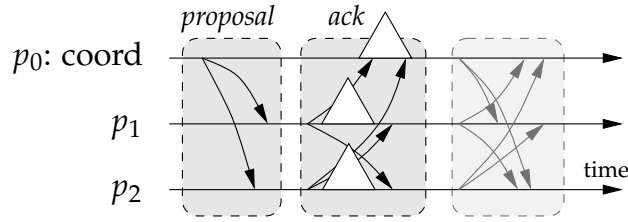


Figure A.2: Communication pattern of the Mostéfaoui-Raynal $\diamond S$ consensus algorithm in runs without failures or wrong suspicions ($n = 3$).

ducing the abort message [Urb03], processes do not start a new round in runs where no suspicions occur, thus reducing the contention in the system in these runs.

In Phase 4, the coordinator waits for a majority ($f + 1$) of answers (lines 31 and 32). If all answers are *acks*, the coordinator reliably broadcasts the decision to all processes (lines 33 and 34). If at least one *nack* is received, the coordinator sends an abort message for round r_p and proceeds to the next round without deciding (line 36). One can show that if $\lceil \frac{n+1}{2} \rceil$ of processes have accepted the coordinator's proposal v , then any future decision is v , although the decision on v might only be taken in a later round.

Finally, whenever a process reliably delivers a decision, it decides if it hasn't already done so before (lines 37 to 40). Figure A.1 illustrates the algorithm in a system with three processes and in a run without failures nor suspicions. The dashed rounded rectangles indicate the communication rounds and the arrows indicate sent messages. Decisions are represented as triangles. The coordinator, p_0 , decides after two communication steps (after receiving the *acks* of p_1 and p_2). Processes p_1 and p_2 decide upon *rdelivering* p_0 's decision, after the third communication step. The fourth communication (greyed out in Figure A.1) is needed by the reliable broadcast algorithm presented in Section A.1.

In runs without failures nor suspicions, the Chandra-Toueg consensus algorithm decides in the first round and thus needs 3 communication steps for all processes to decide (2 steps for the coordinator). To reach this decision, the algorithm needs to send $2n$ point-to-point messages and one reliable broadcast.

A.2.2 Mostéfaoui-Raynal consensus

In [MR99], Mostéfaoui and Raynal present a consensus algorithm based on unreliable failure detectors and quorums. We consider their $\diamond S$ based algorithm here. As for the Chandra-Toueg consensus algorithm, the Mostéfaoui-Raynal algorithm proceeds in rounds and requires a majority of correct processes, but is a decentralized algorithm. The algorithm is presented

Algorithm A.3: Mostéfaoui-Raynal $\diamond\mathcal{S}$ consensus alg. (code of process p)

```

1: procedure propose( $v_p$ )
2:    $estimate_p \leftarrow v_p$  { $p$ 's estimate of the decision value}
3:    $state_p \leftarrow undecided$ 
4:    $r_p \leftarrow 0$  { $p$ 's current round number}
5:   while  $state_p = undecided$  do {rotate through coordinators until decision reached}
6:      $r_p \leftarrow r_p + 1$ 
7:      $c_p \leftarrow (r_p \bmod n) + 1$  { $c_p$  is the current coordinator}
8:      $est\_from\_c_p \leftarrow \perp$  {coordinator's estimate or invalid ( $\perp$ )}
9:     Phase 1: {coordinator proposes new estimate; other processes wait for this new estimate}
10:    if  $p = c_p$  then
11:       $est\_from\_c_p \leftarrow estimate_p$ 
12:    else
13:      wait until received  $(c_p, r_p, est\_from\_c_{c_p})$  from  $c_p$  or  $c_p \in \mathcal{D}_p$  {query F.D.  $\mathcal{D}_p$ }
14:      if received  $(c_p, r_p, est\_from\_c_{c_p})$  from  $c_p$  then { $p$  received  $est\_from\_c_{c_p}$  from  $c_p$ }
15:         $est\_from\_c_p \leftarrow est\_from\_c_{c_p}$ 
16:      send  $(p, r_p, est\_from\_c_p)$  to all
17:    Phase 2: {each process waits for  $\lceil \frac{n+1}{2} \rceil$  replies. If they indicate that  $\lceil \frac{n+1}{2} \rceil$  processes adopted the proposal, the process rbroadcasts a decide message}
18:    wait until for  $\lceil \frac{n+1}{2} \rceil$  processes  $q$  : received  $(q, r_p, est\_from\_c_q)$ 
19:     $rec_p \leftarrow \{est\_from\_c_q \mid p \text{ received } (q, r_p, est\_from\_c_q) \text{ from } q\}$ 
20:    if  $rec_p = \{v\}$  then
21:       $estimate_p \leftarrow v$ 
22:      call takedecision
23:    else if  $rec_p = \{v, \perp\}$  then
24:       $estimate_p \leftarrow v$ 

25: procedure takedecision
26:   decide( $estimate_p$ )
27:    $state_p \leftarrow decided$ 
28:   send  $(p, estimate_p, decide)$  to all

29: when receive  $(q, estimate_q, decide)$  {if  $p$  receives a decide message,  $p$  decides accordingly}
30:   if  $state_p = undecided$  then
31:      $estimate_p \leftarrow estimate_q$ 
32:     call takedecision

```

in Algorithm A.3.

Each round consists of two phases. At the beginning of Phase 1, the coordinator of that round sends its estimate to all processes (line 11, then line 16). Each process then either receives the coordinator's proposal (lines 14 and 15), or suspects the coordinator. In the latter case, the process considers that an invalid value (\perp) was received from the coordinator ($est_from_c_{c_p}$ is initialized to \perp on line 8 and is not modified in the case of a suspicion). In both cases, the process sends the estimate received from the coordinator (a valid value or \perp) to all processes (line 16), which concludes Phase 1 of the algorithm.

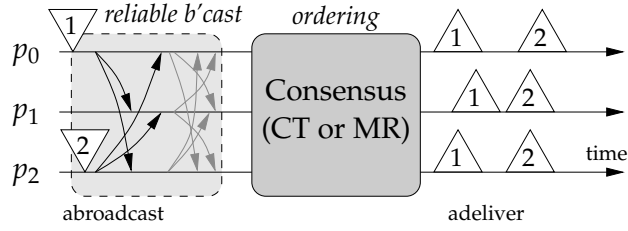


Figure A.3: Communication pattern of the Chandra-Toueg atomic broadcast algorithm ($n = 3$ processes).

In Phase 2, each process waits for a majority of estimates, including the one possibly received from the coordinator (line 18). If all received estimates are the same value v , the process decides v and informs all other processes of its decision (lines 20 to 22). If this is not the case, but at least one received estimate is valid (not \perp), the process sets its own estimate to the received valid estimate and proceeds to the next round (lines 23 and 24). The reliable diffusion of the decision is ensured by an ad-hoc protocol: each process retransmits the decision to all processes upon receiving it for the first time (lines 29 to 32).

The *Uniform agreement* property of consensus is ensured by the fact that if a process p decides v , then p has received the estimate v from $\lceil \frac{n+1}{2} \rceil$ processes. This (and quasi-reliable channels) in turn ensures that all processes have received at least one estimate equal to v and have thus set their own estimate to v . Since the estimates of all processes are equal to v , any subsequent decision can only be v . Figure A.2 illustrates the algorithm in a system with three processes and in a run without failures nor suspicions. In runs without failures nor suspicions, the Mostéfaoui-Raynal consensus algorithm decides in the first round and thus needs 2 communication steps for all processes to decide (1 step for non-coordinator processes in a system with $n = 3$ processes). To reach this decision, the algorithm needs to send $2n^2$ point-to-point messages.

A.3 Atomic broadcast

A.3.1 Chandra-Toueg atomic broadcast

In [CT96], Chandra and Toueg present a reduction of atomic broadcast to consensus. In this thesis, this atomic broadcast algorithm is considered in combination with the Chandra-Toueg and Mostéfaoui-Raynal consensus algorithms presented previously. Furthermore, the Chandra-Toueg atomic broadcast algorithm is slightly modified in that consensus is run on message identifiers instead of the messages themselves, as previously de-

Algorithm A.4: Chandra-Toueg atomic broadcast alg. (code of process p)

```

1: Initialisation:
2:    $received_p \leftarrow \emptyset$  {set of messages received by process  $p$ }
3:    $unordered_p \leftarrow \emptyset$  {set of messages received but not yet ordered by process  $p$ }
4:    $ordered_p \leftarrow \epsilon$  {sequence of identifiers of messages ordered but not yet delivered by  $p$ }
5:    $k \leftarrow 0$  {serial number for consensus executions}

6: procedure  $abroadcast(m)$  {To broadcast a message  $m$  with identifier  $id(m)$ }
7:    $rbroadcast(m)$  to all

8: when  $rdeliver(m)$ 
9:    $received_p \leftarrow received_p \cup \{m\}$ 
10:  if  $id(m) \notin ordered_p$  then
11:     $unordered_p \leftarrow unordered_p \cup \{id(m)\}$ 

12: when  $unordered_p \neq \emptyset$  {a consensus is run whenever there are unordered messages}
13:    $k \leftarrow k + 1$ 
14:    $propose(k, unordered_p, rcv)$  { $k$  distinguishes independent consensus executions}
15:   wait until  $decide(k, idSet^k)$ 
16:    $unordered_p \leftarrow unordered_p \setminus idSet^k$ 
17:    $idSeq^k \leftarrow$  elements of  $idSet^k$  in some deterministic order
18:    $ordered_p \leftarrow ordered_p \triangleright idSeq^k$ 

19: {delivers messages ordered and received}
20: when  $ordered_p \neq \emptyset$  and  $\exists m \in received_p$  such that  $head.ordered_p = id(m)$ 
21:    $adeliver(m)$ 
22:    $ordered_p \leftarrow tail.ordered_p$ 

```

scribed in [Urb03]. This reduction of atomic broadcast to consensus on message identifiers is correct if we assume that the underlying quasi-reliable channels offer first-in first-out delivery of messages. If channels are not first-in first-out, the reduction is trickier, as discussed in Chapter 5. The atomic broadcast algorithm is presented in Algorithm A.4.

Whenever a message m is *abroadcast*, it is reliably broadcast to all processes (line 7, see also Figure A.3, *reliable b'cast*). When a process p *rdelivers* m , m is added to its set of received messages (lines 8 and 9). Furthermore, if the identifier of m is not in the sequence of ordered identifiers, p adds the identifier of m to the set of unordered identifiers (lines 10 and 11).

When p 's set of unordered identifiers is not empty, p proposes this set for the next consensus execution (lines 12 to 14, see also Figure A.3, *ordering*). p then waits until consensus terminates, removes the decision (a set of message identifiers) from the set of unordered message identifiers and adds the decision to the sequence of ordered message identifiers (lines 15 to 18).

Finally, the *abroadcast* messages are *adelivered* by p following the order of their identifiers in the $ordered_p$ sequence (lines 20 to 22). Figure A.3 shows

the communication pattern of the atomic broadcast algorithm: an *abroadcast* message is first reliably broadcast to all processes and consensus is then used to decide the order of the messages.

The Chandra-Toueg atomic broadcast algorithm needs one reliable broadcast followed by a consensus to *adeli*ver a message. The number of communication steps and sent messages thus depend on the underlying algorithms used by atomic broadcast.

If the reliable broadcast and Chandra-Toueg consensus algorithms above are used and there are no failures nor suspicions, then 4 communication steps and $2n(n + 1)$ messages are necessary for all processes to *adeli*ver a message. If Mostéfaoui-Raynal's consensus algorithm is used instead, then 3 communication steps and $3n^2$ messages are needed to *adeli*ver a message.

A.3.2 Moving sequencer uniform atomic broadcast

The moving sequencer algorithm is a group membership and token based uniform atomic broadcast algorithm. The algorithm considered in this thesis is based on work in [CM84, CM95, CMA97, DSU04, WMK94] and is slightly adapted to reduce the latency of *abroadcasting* and *adeli*vering a message. Furthermore, the algorithm requires that at least $\lceil \frac{n+1}{2} \rceil$ processes remain correct in each view (with n the number of processes in the view).

In contrast with the other algorithms studied in this thesis which use failure detectors, the moving sequencer algorithm does not tolerate failures directly, but instead relies on an underlying group membership protocol to handle view changes triggered by suspicions and crashes of processes. As a consequence, the algorithm is expressed in two parts: Section A.3.2.A presents the moving sequencer algorithm when no processes crash or are suspected (and the view of the group does not change). Suspicions and crashes are handled by the group membership protocol presented in Algorithm A.6 (Section A.3.2.B) and inspired from the group membership algorithm in [USS03]. Since the performance of the moving sequencer algorithm is only studied in a system without process failures or suspicions (in Section 7.3.2), Algorithm A.6 is only presented here for the sake of completeness.

A.3.2.A Atomic broadcast algorithm

Algorithm A.5 presents the moving sequencer atomic broadcast algorithm for a process p . Whenever a message is *abroadcast*, it is sent to all processes (lines 12 and 13, also illustrated in Figure A.4, *send to sequencer*) so that it can be ordered by the moving sequencer. Upon receiving a message m (line 14), process p adds m to the set of received messages $received_p$ (line 15) and then adds m to its set of unordered messages if m is not already ordered (lines 16 and 17).

Algorithm A.5: Moving sequencer uniform atomic broadcast algorithm
(code of process p)

```

1: Initialisation:
2:    $received_p \leftarrow \emptyset$  {set of received messages (receive queue)}
3:    $ordered_p \leftarrow \emptyset$  {set of messages with a seq. number}
4:    $unordered_p \leftarrow \emptyset$  {set of unordered messages}
5:    $nextdeliver_p \leftarrow 0$  {sequence number of the next message batch to adelivered }
6:    $nextstable_p \leftarrow 0$  {sequence number of the next stable (garbage collected) message batch}
7:    $acks_p \leftarrow (\emptyset, \dots, \emptyset)$  {array  $[p_0, \dots, p_{n-1}]$  of seq. num. sets (acknowledged batches)}
8:    $toknext_p \leftarrow$  successor of  $p$  in token ring {identity of the next process along the logical ring}
9:    $toksender_p \leftarrow$  process with smallest index in  $\Pi$  {sender process of the token}
10:  if  $p = toksender_p$  then {virtual message to initiate the token rotation}
11:    send  $(\emptyset, -1, acks_p)$  to all {format: message set, seq. number, acks}

12: procedure  $abroadcast(m)$  {To abroadcast a message  $m$ }
13:  send  $m$  to all

14: when receive  $m$ 
15:    $received_p \leftarrow received_p \cup \{m\}$  {add  $m$  to received messages}
16:   if  $m \notin \bigcup_{(msgs, -) \in ordered_p} msgs$  then
17:      $unordered_p \leftarrow unordered_p \cup \{m\}$  { $m$  is not ordered yet}

18: when receive  $(msgs, seqnum, acks)$  from  $toksender_p$  {token reception}
19:    $toksender_p \leftarrow$  successor of  $toksender_p$  in the token ring
20:    $acks_p \leftarrow acks_p \cup acks$ 
21:   if  $msgs \neq \emptyset$  then {token contains batch of ordered messages}
22:      $ordered_p \leftarrow ordered_p \cup \{(msgs, seqnum)\}$ 
23:      $unordered_p \leftarrow unordered_p \setminus msgs$ 
24:     for all  $(msgs', seq') \in ordered_p$  s.t.  $msgs' \subseteq received_p$  do {Ack received messages}
25:        $acks_p[p] \leftarrow acks_p[p] \cup \{seq'\}$ 
26:     if  $p = toksender_p$  then
27:       wait until  $unordered_p \neq \emptyset \vee ordered_p \neq \emptyset$ 
28:       if  $unordered_p \neq \emptyset$  then {If unordered messages exist, send them in the token}
29:         send  $(unordered_p, seqnum + 1, acks_p)$  to all
30:       else {Else, send token with same seqnum as before}
31:         send  $(\emptyset, seqnum, acks_p)$  to all
{adeliivers messages ordered and received}

32: when  $\exists (msgs, nextdeliver_p) \in ordered_p$  s.t.  $msgs \subseteq received_p$  and
 $\exists \Pi_{safe} \subseteq \Pi : (|\Pi_{safe}| > \frac{n}{2} \text{ and } \forall q \in \Pi_{safe} : nextdeliver_p \in acks_p[q])$ 
33:   adeliiver messages in  $msgs$  in some deterministic order
34:    $nextdeliver_p \leftarrow nextdeliver_p + 1$ 
{Garbage collects ordered messages received by all processes}

35: when  $nextstable_p < nextdeliver_p$  and
 $\exists (msgs, nextstable_p) \in ordered_p$  s.t.  $\forall q \in \Pi : nextstable_p \in acks_p[q]$ 
36:    $ordered_p \leftarrow ordered_p \setminus \{(msgs, nextstable_p)\}$ 
37:    $received_p \leftarrow received_p \setminus msgs$ 
38:    $nextstable_p \leftarrow nextstable_p + 1$ 

```

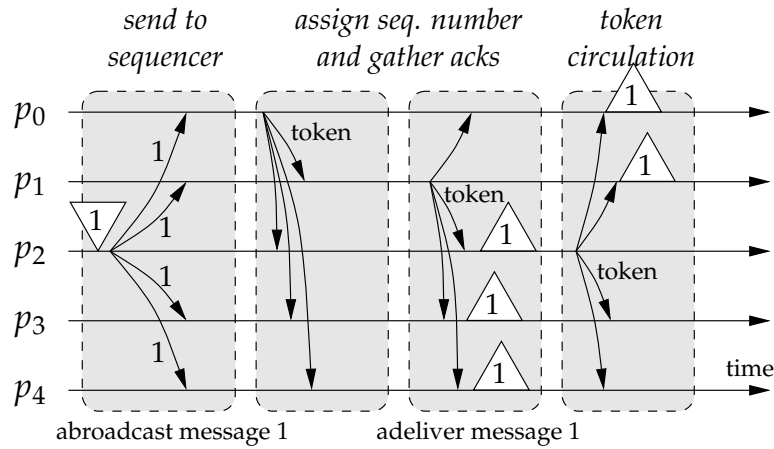


Figure A.4: Communication pattern of the Moving Sequencer atomic broadcast algorithm ($n = 5$ processes).

Token circulation The token circulation is handled in lines 18 to 31 (and the initial token is sent on line 11). The token contains a set of messages to be ordered (or an empty set if no such messages exist), a tentative sequence number assigned to this set and finally a set of acknowledgments. Furthermore, p receives the token from the process $toksender_p$ (which is updated each time a token is received, at line 19).

Upon receiving the token, p incorporates the acknowledgments transported in the token in its own data structure $acks_p$ (line 20). Then, if the token contains a non-empty set of ordered messages (lines 21 to 23), the pair (messages, sequence number) is added to $ordered_p$ (the set of ordered message batches). The set of unordered messages is then updated (line 23). In lines 24 and 25, p updates its set of acknowledgments: all message batches that are ordered and that have been received are acknowledged.

Finally, if p is the next token sender (line 26), it waits until either $unordered_p$ (the set of unordered messages) or $ordered_p$ (the set of ordered message batches) is non-empty (line 27). In case these two sets are both empty, the token circulation is suspended, as all processes have already *adelivered* all the *abroadcast* messages. After that, if the set of unordered messages is not empty, a token is sent with the unordered messages and a new sequence number (lines 28 and 29, see also Figure A.4, *assign seq. num. and gather acks*). Otherwise, a token with an empty message set and the last sequence number is sent (line 31).

adelivering and garbage collecting messages The *adelivery* of messages is handled in lines 32 to 34. Whenever the messages ordered at position $nextdeliver_p$ have been received and acknowledged by at least $\lceil \frac{n+1}{2} \rceil$ processes (line 32), these messages are *adelivered* in a deterministic order

(line 33) and $nextdeliver_p$ is incremented (line 34).

Lines 35 to 38 garbage collect ordered messages. The condition on line 35 ensures that only *adelivered* messages are garbage collected ($nextstable_p < nextdeliver_p$) and that these messages have been acknowledged by all processes ($\forall q \in \Pi : nextstable_p \in acks_p[q]$). The garbage collected messages are removed from $ordered_p$ (line 36) and $received_p$ (line 37). Finally, $nextstable_p$ is incremented.

The garbage collection is not necessary for the correctness of the algorithm but is presented here since in the Chandra-Toueg atomic broadcast algorithm, *adelivered* messages do not need to be stored in memory.

Complexity of the algorithm The moving sequencer atomic broadcast algorithm needs $1 + \lceil \frac{n+1}{2} \rceil$ communication steps for all processes to *adeliver* an *abroadcast* message m (the first processes *adeliver* m after $1 + f = 1 + \lfloor \frac{n-1}{2} \rfloor$ steps). Furthermore, a single *abroadcast* message m results in $n \cdot (1 + \lceil \frac{n+1}{2} \rceil)$ point-to-point messages before all processes *adeliver* m .

A.3.2.B Group membership algorithm

Algorithm A.6 presents the group membership protocol for the moving sequencer atomic broadcast algorithm. It is inspired from the work in [USS03].

Whenever process p suspects another processes or that p receives a message indicating that a suspicion has occurred on another process (line 3), the view change protocol is initiated. Process p starts by iterating over all its ordered message batches (line 4): all messages that have been ordered but not received are removed (line 5). Process p then sends its state to everyone (line 6); the state includes the set of received messages $received_p$, all the ordered messages it knows of and that it has received ($ordered_p$) as well as the current view number.

Then, process p waits until the state of at least $\lceil \frac{n+1}{2} \rceil$ processes in view $viewnumber_p$ have been received (line 7). It merges the sets received from other processes with its own (lines 8 and 9) and then proposes a value for consensus (line 10). The proposed value is a tuple containing the new proposed view ($\Pi \setminus \{suspected\ processes\}$: the current view, minus the suspected processes), the ordered messages p knows of, as well as the set of received messages.

After a decision is taken on a tuple containing the new view Π_{new} , the sets of ordered messages $orderedAll$ and received messages $receivedAll$ (line 11), p adds the sets to its local variables (lines 13 and 14). Furthermore, on line 12, the highest sequence number in $orderedAll$ is stored in $maxdeliver$ (its use is explained below).

At this point in the protocol, $received_p$ (the set of received messages) can contain messages that have already been *adelivered* by p . Lines 15 to

Algorithm A.6: Group membership protocol for the moving sequencer algorithm (code of process p)

```

1: Initialisation:
2:    $viewnumber_p \leftarrow 0$  {number of the current view}

3: upon suspicion or receive  $(r, o, viewnumber_p)$  do
4:   for all  $(msgs, seq) \in ordered_p$  s.t.  $msgs \not\subseteq received_p$  do
5:      $ordered_p \leftarrow ordered_p \setminus \{(msgs, seq)\}$  {Remove non-received ordered messages}
6:   send  $(received_p, ordered_p, viewnumber_p)$  to all
7:   wait until received  $(r, o, viewnumber_p)$  from  $\lceil \frac{n+1}{2} \rceil$ 
{Update the local data structures with those received from other processes}
8:    $received_p \leftarrow received_p \cup (\bigcup_{received(r, -, viewnumber_p)} r)$ 
9:    $ordered_p \leftarrow ordered_p \cup (\bigcup_{received(-, o, viewnumber_p)} o)$ 
{Decide on the new group and the set of adelivered messages}
10:  propose  $(\Pi \setminus \{suspected\ processes\}, ordered_p, received_p)$ 
11:  wait until decide  $(\Pi_{new}, orderedAll, receivedAll)$ 
12:   $maxdeliver \leftarrow \max_{seqnum}(-, seqnum) \in orderedAll$ 
13:   $ordered_p \leftarrow ordered_p \cup orderedAll$ 
14:   $received_p \leftarrow received_p \cup receivedAll$ 
{Remove messages that have already been adelivered }
15:  while  $\exists (msgs, seq') \in ordered_p$  s.t.  $seq' < nextdeliver_p$  do
16:     $received_p \leftarrow received_p \setminus msgs$ 
{adeliver new messages if possible}
17:  while  $\exists (msgs, nextdeliver_p) \in ordered_p$  and  $nextdeliver_p \leq maxdeliver$  do
18:    adeliver messages in  $msgs$  in some deterministic order
19:     $received_p \leftarrow received_p \setminus msgs$ 
20:     $nextdeliver_p \leftarrow nextdeliver_p + 1$ 
{reset the data structures and install the new view}
21:   $ordered_p \leftarrow \emptyset$ 
22:   $nextstable_p \leftarrow nextdeliver_p$ 
23:   $acks_p \leftarrow (\emptyset, \dots, \emptyset)$ 
24:   $viewnumber_p \leftarrow viewnumber_p + 1$ 
25:   $\Pi \leftarrow \Pi_{new}$ 
26:   $toksender_p \leftarrow$  process with smallest index in  $\Pi$ 
27:  if  $p = toksender_p$  then {virtual message to initiate the token rotation}
28:    send  $(\emptyset, nextdeliver_p - 1, acks_p)$  to all

```

16 remove all such messages from $received_p$ (to avoid reordering them and *adelivering* them twice). After line 16, $received_p$ thus only contains messages that have not yet been *adelivered*. The protocol continues (lines 17 to 20) by *adelivering* all messages in $ordered_p$ with a sequence number that is greater than $nextdeliver_p$ and smaller than the value of $maxdeliver$ (which is the same for all processes that participated in the consensus on lines 10 and 11).

The $maxdeliver$ limit is necessary to avoid the following scenario: consider that a single process p receives the token (with a message m and a sequence number s) from a process q that crashes shortly thereafter. The group membership protocol starts running, but p 's $ordered_p$ set (which contains the (m, s) pair) isn't received by any other process (all processes only wait for the state message of $\lceil \frac{n+1}{2} \rceil$ processes). Furthermore, the consensus decision is not p 's proposal. At this point, no process, except p , has m in its set of ordered messages. To ensure the uniform agreement of atomic broadcast, p should thus *not adeliver* m . This is done by preventing the *adelivery* of messages with a sequence number greater than $maxdeliver$ (which is the same for all processes and strictly smaller than s in this example).

Finally, lines 21 to 28 reinitialize the state of p . Notice that $ordered_p$ is reset to the empty set. Thus, some messages that were ordered in view i might be reordered in view $i + 1$. Such messages however, although ordered in view i , are never *adelivered* in view i and the specification of atomic broadcast thus holds.

Modeling and validating the performance of atomic broadcast algorithms in high latency networks

B.1 Analytical performance in a wide area network with three locations

The following section presents the analytical performance of *CT*, *MR* and *TokenFD* in the wide area network model with three locations. The first part presents the derivation of the results of *CT* and *MR* using a fixed initial coordinator. The slightly more complex case of *TokenFD* or *CT* and *MR* using a shifting initial coordinator is then discussed.

B.1.1 Chandra-Toueg atomic broadcast

B.1.1.A Fixed initial coordinator

Table B.1 presents the average latency of Chandra-Toueg's atomic broadcast algorithm using the *CT* and *MR* consensus algorithms in a wide area network with three locations and a fixed initial coordinator. In the following paragraphs, we show how to derive the results for an initial coordinator that is always located on location p_0 . The results for an initial coordinator on location p_1 or p_2 are derived similarly.

Phase (1): Message diffusion Let p_0 be the initial coordinator of the *CT* or *MR* consensus algorithm. Any process that *abroadcasts* a message, needs to send it to p_0 in phase (1). The cost of this diffusion is negligible

Table B.1: Average latency to *adeli*ver a message in the three-location wide area network model, using Chandra-Toueg’s algorithm (with *CT* and *MR*’s consensus algorithm and a fixed initial coordinator).

(a) <i>CT</i> consensus			
	0	1	2
coord. proc.:	0	1	2
(1) diffusion:	$\frac{d_0+d_2}{3}$	$\frac{d_0+d_1}{3}$	$\frac{d_2+d_1}{3}$
(2) waiting:	d_2	d_1	d_2
(3) ordering:	$\frac{7d_2+d_0}{3}$	$\frac{7d_1+d_0}{3}$	$\frac{7d_2+d_1}{3}$
Latency:	$\frac{2d_0+11d_2}{3}$	$\frac{2d_0+11d_1}{3}$	$\frac{2d_1+11d_2}{3}$
(b) <i>MR</i> consensus			
	0	1	2
coord. proc.:	0	1	2
(1) diffusion:	$\frac{d_0+d_2}{3}$	$\frac{d_0+d_1}{3}$	$\frac{d_2+d_1}{3}$
(2) waiting:	d_2	d_1	d_2
(3) ordering:	$\frac{3d_2+d_0}{3}$	$\frac{3d_1+d_0}{3}$	$\frac{3d_2+d_1}{3}$
Latency:	$\frac{2d_0+7d_2}{3}$	$\frac{2d_0+7d_1}{3}$	$\frac{2d_1+7d_2}{3}$

if p_0 *abroadcasts* a message and equal to d_0 and d_2 , if p_1 , respectively p_2 are the *abroadcasters* of the message. On average (and since all three processes *abroadcast* at the same rate), we have a diffusion cost of $\frac{d_0+d_2}{3}$.

Phase (2): Waiting phase In phase (2), a message sent to the coordinator has to wait until a new consensus execution can start (assuming that a consensus execution is already underway). The coordinator p_0 can decide after two communication steps (with any of the two other processes) and thus starts a new consensus execution after that. The duration of these two communication steps is either $2d_0$ or $2d_2$ and since $d_0 \geq d_2$, the coordinator can decide after $2d_2$. Thus, a new consensus execution is started on average each $2d_2$ time units. Consequently, the average waiting time for an *abroadcast* message is d_2 time units.

Phase (3): Ordering (Consensus) Finally, the cost of phase (3) is the following. In the *CT* consensus algorithm, as seen previously, the coordinator decides after $2d_2$ time units and sends its decision to the two other processes. The process on location 1 thus decides d_0 time units after the coordinator whereas the process on location 2 decides d_2 time units after the coordinator. On average, this gives us a cost of phase (3) equal to $\frac{(2d_2)+(2d_2+d_0)+(2d_2+d_2)}{3} = \frac{7d_2+d_0}{3}$ for the *CT* consensus algorithm.

In the *MR* consensus algorithm, the coordinator also decides after $2d_2$ time units. However, if $n = 3$, the two other processes p_1 and p_2 can decide

as soon as they get the coordinator's initial proposal, after d_0 and d_2 time units respectively. On average, this gives us a cost of phase (3) equal to $\frac{(2d_2)+(d_0)+(d_2)}{3} = \frac{3d_2+d_0}{3}$ for the *MR* consensus algorithm.

The sum of the three phases, which is the average latency for *abroadcasting* a message using Chandra-Toueg's atomic broadcast algorithm is thus equal to $\frac{11d_2+2d_0}{3}$ time units if *CT*'s consensus algorithm is used and $\frac{7d_2+2d_0}{3}$ if *MR*'s consensus algorithm is used.

B.1.1.B Shifting coordinator

The case of an initial coordinator that changes between consensus execution k and $k + 1$ is slightly more difficult to analyze. Tables B.2 and B.3 present the $CostSend_{i,j}$, $CostWait_{i,j}$, $CostOrder_j$ and $OrderedBy_{i,j}$ matrices in our modeled system where the *CT*, respectively the *MR*, consensus algorithms are used and the initial coordinator changes at each consensus execution. The average latency of atomic broadcast is then easily derived using the results presented in Section 8.4.1 (but its expression is too complex to be explicitly presented here).

B.1.2 *TokenFD* atomic broadcast

Table B.4 presents the $CostSend_{i,j}$, $CostWait_{i,j}$, $CostOrder_j$ and $OrderedBy_{i,j}$ matrices in our modeled system with the *TokenFD* atomic broadcast algorithm. The average latency of atomic broadcast is then easily derived following the results presented in Section 8.4.1 (again, its expression is too complex to be explicitly presented here).

Table B.2: $CostSend_{i,j}$, $CostWait_{i,j}$, $CostOrder_j$ and $OrderedBy_{i,j}$ in the three-location wide area network model, using Chandra-Toueg's algorithm (with CT's consensus algorithm and a shifting initial coordinator).

$$\begin{array}{l}
 \text{(a) } CostSend_{i,j} \\
 \begin{pmatrix} 0 & d_0 & d_2 \\ d_0 & 0 & d_1 \\ d_2 & d_1 & 0 \end{pmatrix} \\
 \text{(b) } CostWait_{i,j} \\
 \begin{pmatrix} 2d_2 & \max(\frac{3d_2+d_1-d_0}{2}, 0) & d_2 \\ d_2 & 2d_2 + \frac{d_0}{2} + \min(\frac{d_0}{2}, \frac{3d_2+d_1}{2}) & \frac{3d_2+d_0-d_1}{2} \\ \frac{d_0+3d_1-d_2}{2} & d_1 & 2d_1 \end{pmatrix} \\
 \text{(c) } CostOrder_j \\
 \begin{pmatrix} \frac{d_0+7d_2}{3} \\ \frac{d_0+7d_1}{3} \\ \frac{d_1+7d_2}{3} \end{pmatrix} \\
 \text{(d) } OrderedBy_{i,j} \\
 \frac{1}{3 \cdot (d_0+3d_1+5d_2)} \cdot \begin{pmatrix} 4d_2 & \max(3d_2 + d_1 - d_0, 0) & 2d_2 \\ 2d_2 & 2d_2 + d_0 + \min(d_0, 3d_2 + d_1) & 3d_2 + d_0 - d_1 \\ d_0 + 3d_1 - d_2 & 2d_1 & 4d_1 \end{pmatrix}
 \end{array}$$

 Table B.3: $CostSend_{i,j}$, $CostWait_{i,j}$, $CostOrder_j$ and $OrderedBy_{i,j}$ in the three-location wide area network model, using Chandra-Toueg's algorithm (with MR's consensus algorithm and a shifting initial coordinator).

$$\begin{array}{l}
 \text{(a) } CostSend_{i,j} \\
 \begin{pmatrix} 0 & d_0 & d_2 \\ d_0 & 0 & d_1 \\ d_2 & d_1 & 0 \end{pmatrix} \\
 \text{(b) } CostWait_{i,j} \\
 \begin{pmatrix} d_2 & \max(\frac{d_1+d_2-d_0}{2}, 0) & 0 \\ 0 & \min(d_0, \frac{d_0+d_1+d_2}{2}) & \frac{d_0+d_2-d_1}{2} \\ \frac{d_0+d_1-d_2}{2} & 0 & d_1 \end{pmatrix} \\
 \text{(c) } CostOrder_j \\
 \begin{pmatrix} \frac{d_0+3d_2}{3} \\ \frac{d_0+3d_1}{3} \\ \frac{d_1+3d_2}{3} \end{pmatrix} \\
 \text{(d) } OrderedBy_{i,j} \\
 \frac{1}{3 \cdot (d_0+d_1+d_2)} \cdot \begin{pmatrix} 2d_2 & \max(d_2 + d_1 - d_0, 0) & 0 \\ 0 & d_0 + \min(d_0, d_1 + d_2) & d_0 + d_2 - d_1 \\ d_0 + d_1 - d_2 & 0 & 2d_1 \end{pmatrix}
 \end{array}$$

Table B.4: $CostSend_{i,j}$, $CostWait_{i,j}$, $CostOrder_j$ and $OrderedBy_{i,j}$ in the three-location wide area network model, using the *TokenFD* atomic broadcast algorithm.

$$\begin{array}{ll}
 \text{(a) } CostSend_{i,j} & \text{(b) } CostWait_{i,j} \\
 \begin{pmatrix} 0 & d_0 & d_2 \\ d_0 & 0 & d_1 \\ d_2 & d_1 & 0 \end{pmatrix} & \begin{pmatrix} d_2 & \max(\frac{d_1+d_2-d_0}{2}, 0) & 0 \\ 0 & \min(d_0, \frac{d_0+d_1+d_2}{2}) & \frac{d_2+d_0-d_1}{2} \\ \frac{d_0+d_1-d_2}{2} & 0 & d_1 \end{pmatrix} \\
 \\
 \text{(c) } CostOrder_j & \\
 \begin{pmatrix} \frac{4d_0+d_1}{3} \\ \frac{4d_1+d_2}{3} \\ \frac{4d_2+d_0}{3} \end{pmatrix} & \\
 \\
 \text{(d) } OrderedBy_{i,j} & \\
 \frac{1}{3 \cdot (d_0+d_1+d_2)} \cdot \begin{pmatrix} 2d_2 & \max(d_1+d_2-d_0, 0) & 0 \\ 0 & \min(2d_0, d_0+d_1+d_2) & d_0+d_2-d_1 \\ d_0+d_1-d_2 & 0 & 2d_1 \end{pmatrix} &
 \end{array}$$

B.2 Analytical performance in a wide area network with two locations

The following paragraphs present the analytical performance of the *TokenFD* atomic broadcast algorithm and the Chandra-Toueg atomic broadcast algorithm using either the *CT* or the *MR* consensus algorithm in the two-location wide area network model. The results presented here can either be derived directly, or from the results in Appendix B.1 by replacing d_0 and d_1 by D and d_2 by 0 (p_0 and p_2 are in the local location and p_1 is in the distant location). In the following paragraphs, we show how to derive these results directly.

Table B.5 summarizes the latency of *abroadcast* using the *TokenFD* algorithm or the Chandra-Toueg atomic broadcast algorithm. In the case of the Chandra-Toueg algorithm, the table presents results for the *CT* (Table B.5(a)) and *MR* consensus (Table B.5(b)) algorithms in the case of an initial coordinator that is fixed (on the local or distant location) or that changes at each new consensus execution.

Table B.5: Average latency to *adeli*ver a message in the two-location wide area network model, using Chandra-Toueg’s algorithm (with *CT* or *MR*’s consensus algorithm) or the *TokenFD* algorithm. Results are given for an initial coordinator (*MR*, *CT*) on a local location, on the distant location or that shifts at each new consensus execution.

	(a) <i>CT</i> consensus			(b) <i>MR</i> consensus		
coord. loc.:	local	distant	shift.	local	distant	shift.
(1) diffusion:	$\frac{D}{3}$	$\frac{2D}{3}$	$\frac{D}{6}$	$\frac{D}{3}$	$\frac{2D}{3}$	0
(2) waiting:	0	D	$\frac{5D}{3}$	0	D	D
(3) ordering:	$\frac{D}{3}$	$\frac{8D}{3}$	$\frac{13D}{18}$	$\frac{D}{3}$	$\frac{4D}{3}$	$\frac{2D}{3}$
Latency:	$\frac{2D}{3}$	$\frac{13D}{3}$	$\frac{23D}{9}$	$\frac{2D}{3}$	3D	$\frac{5D}{3}$

	(c) <i>TokenFD</i>
coord. loc.:	N/A
(1) diffusion:	0
(2) waiting:	D
(3) ordering:	$\frac{2D}{3}$
Latency:	$\frac{5D}{3}$

B.2.1 Chandra-Toueg atomic broadcast

As previously discussed in Section 8.4.1, the cost of the three phases depends on the choice of the privileged process in the ordering phase (3), *i.e.* the coordinator process in the *CT* and *MR* consensus algorithms. Three cases are considered : the initial coordinator (1) is always on a local location, (2) is always on the distant location or (3) shifts from location to location at each new consensus execution. Table B.5 summarizes the latency of *abroadcast* in all three cases, when the *CT* (Table B.5(a)) and *MR* (Table B.5(b)) consensus algorithms are used. The analytical latencies of the cases where the initial coordinator is on a fixed location or shifts at each consensus execution are derived in the following paragraphs.

B.2.1.A Fixed coordinator

Phase (1): Message diffusion The cost of the message diffusion phase is the following: (1) if the initial coordinator is on a local location, it receives *abroadcast* messages from itself and its local peer (with a negligible cost) and from the distant location (with a cost of D). The average cost for diffusing the message to the local coordinator is therefore $\frac{D}{3}$. (2) If the coordinator is on the distant location, it receives messages from the local locations (both with a cost of D) and from itself (with a negligible cost). The average cost for diffusing the message to the distant initial coordinator is thus $\frac{2D}{3}$.

Phase (2): Waiting phase The duration of the waiting phase is the following. In the *CT* consensus algorithm, the coordinator decides after 2 communication steps and atomic broadcast immediately starts a new consensus (if unordered messages are waiting), as illustrated in Figures B.1(a) and B.1(b) (the consensus execution II starts before all processes have finished consensus execution I). The cost of these two communication steps is negligible if the coordinator is on the local location (thus, the waiting time for messages to be proposed in a consensus is negligible). If the coordinator is distant, the two communication steps take $2D$ time units and the messages wait on average D time units to be proposed in a consensus (since messages are *abroadcast* following a Poisson process). The reasoning (and the average waiting times) for the *MR* consensus algorithm is similar, since the coordinator again needs two communication steps to reach a decision.

Phase (3): Ordering (Consensus) Finally, the cost of the *CT* consensus phase is illustrated in Figures B.1(a) and B.1(b) and is the following. The coordinator can decide after two communication steps and all other processes after three steps. (1) If the coordinator is on the local location, a decision is taken after two local communication steps (with a negligible cost) and is received by the other local location one local communication

step later (again, with a negligible cost). The distant location receives the coordinator's decision after one distant communication step (with a cost of D). The average latency is thus $\frac{D}{3}$. (2) If the coordinator is on the distant location, it needs 2 distant communication steps to decide (with a cost of $2D$), whereas both local locations decide one distant communication step later (with a total cost of $3D$ for both local locations). The average decision latency with a distant coordinator is therefore $\frac{8D}{3}$.

The cost of the consensus phase using the *MR* algorithm is similar. The case of a local coordinator gives the same result as *CT* with an average latency of $\frac{D}{3}$. In the case of a distant coordinator, both local locations decide as soon as they receive the coordinator's proposal and their own acknowledgment (resulting in a latency of D). The coordinator decides as soon as it gets an acknowledgment from a local location, after a total time of $2D$. The average latency over all processes is therefore $\frac{4D}{3}$.

B.2.1.B Shifting coordinator

When the initial coordinator changes at each new consensus execution, the analysis is slightly more complex than in the case of an initial coordinator that remains on a single location (presented in Appendix B.2.1.A). Indeed, the consensus executions no longer have the same duration, which in turn means that the messages that are *abroadcast* aren't uniformly distributed among the different consensus executions. For example, if short and long consensus executions alternate, then more messages are ordered in the short consensus (since more unordered messages are accumulated during the execution of the long consensus). A more precise end-to-end analysis of the latencies of the *abroadcast* messages is necessary and is presented in the following paragraphs for the case of the *CT* consensus algorithm. The results with the *MR* consensus algorithm have been derived similarly.

Phase (1): Message diffusion Figure B.1(c) presents the sequence of *CT* consensus executions when the initial coordinator changes each time. Two consensus executions with an initial coordinator on the local location closely follow each other, followed by an execution with a coordinator on the distant location. We start by analyzing which messages are ordered in which consensus executions.

A message m *abroadcast* by p_2 or p_0 (the two processes on the local location) is (almost) always proposed in consensus executions where p_2 is the coordinator. Indeed, when m reaches p_1 , either p_1 has already started the consensus in which it is coordinator (and cannot add m to that consensus) or m is being decided upon in a consensus where p_2 is the coordinator. A message m *abroadcast* by p_1 between t_1 and t_3 (see Figure B.1(c)) is ordered in a consensus execution with p_1 as a coordinator. Messages *abroadcast* by

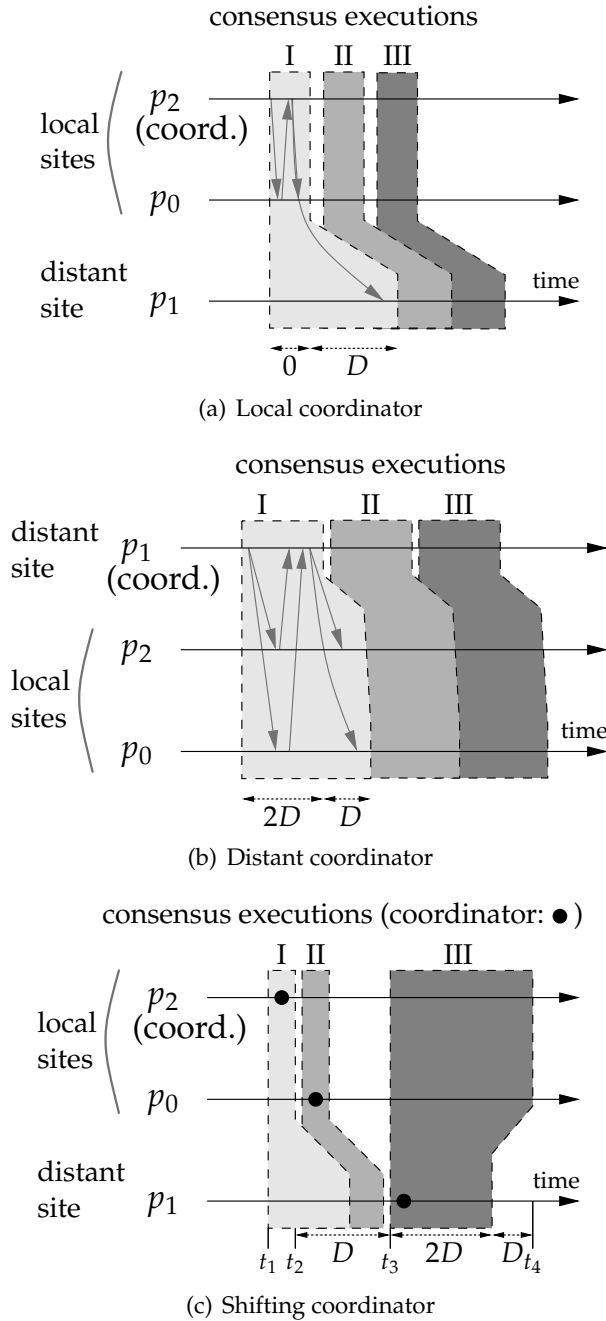


Figure B.1: Execution pattern of the Chandra-Toueg consensus algorithm in the two-location wide area network model and in the case of a coordinator on a local location, a distant location or shifting between locations at each execution.

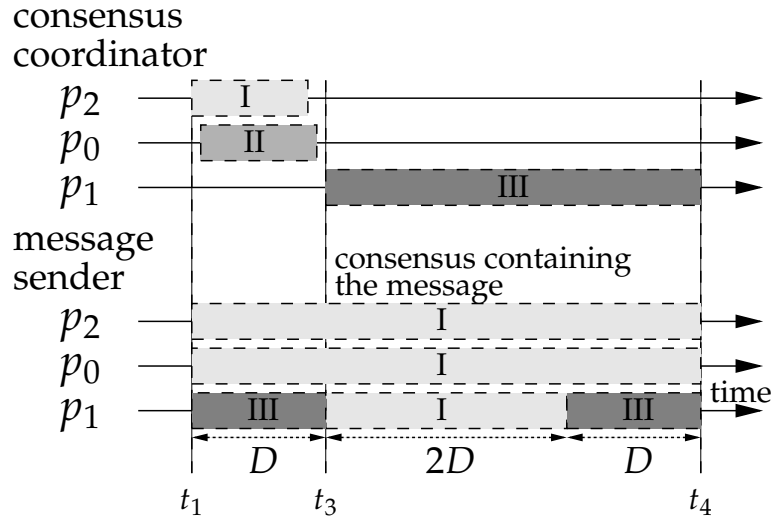


Figure B.2: Messages *abroadcast* by p_0 or p_2 are all ordered in consensus executions with p_2 as coordinator (noted I). Messages *abroadcast* by p_1 are ordered in executions with p_1 or p_2 as coordinator (noted III and I respectively).

p_1 between t_3 and $t_3 + 2D$ are received by p_2 and p_0 before t_4 and are thus ordered in a consensus execution with p_2 as coordinator. Finally, messages *abroadcast* between $t_3 + 2D$ and t_4 do not reach p_2 before t_4 and are later ordered in a consensus execution with p_1 as coordinator.

In total, $\frac{5}{6}$ of all messages are ordered in consensus executions with p_2 as coordinator and the remaining $\frac{1}{6}$ when p_1 is coordinator (consensus executions with p_0 as coordinator order only a negligible amount of messages), as summarized in Figure B.2. The sending time of $\frac{5}{6}$ of the messages is thus 0, whereas it is equal to D in $\frac{1}{6}$ of the cases. The average sending time over all messages is thus $\frac{D}{6}$.

Phase (2): Waiting phase On average, the messages *abroadcast* by p_0 and p_2 ($\frac{2}{3}$ of the messages) wait $2D$ time units before being proposed in a consensus (with p_2 as a leader). Among the messages *abroadcast* by p_1 ($\frac{1}{3}$ of all messages), the waiting time is on average D . The average waiting time over all messages is thus $\frac{5}{3}D$.

Phase (3): Ordering (Consensus) As presented in Section B.2.1.A, the consensus executions where p_2 (on the local location) is the coordinator, the average latency is $\frac{D}{3}$. These executions order $\frac{5}{6}$ of all messages. The remaining $\frac{1}{6}$ of all messages are ordered in executions with p_1 (on the distant location) as coordinator, and thus with an average latency of $\frac{8D}{3}$. The

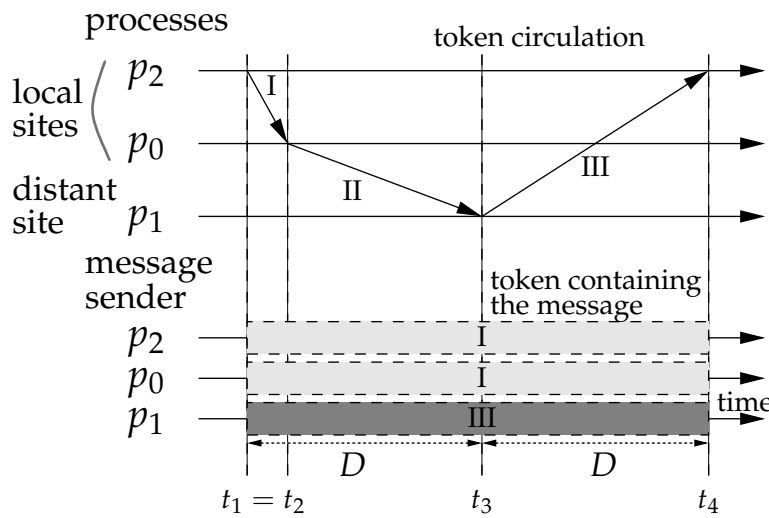


Figure B.3: Token circulation (top) in the two-location wide area network model and a presentation of which token contains the messages *abroadcast* by the three processes (bottom).

average consensus execution time over all messages is thus $\frac{13D}{18}$.

Summary Globally, this gives an average latency of $\frac{23D}{9}$ for *abroadcasting* a message using the Chandra-Toueg atomic broadcast and consensus algorithms if the coordinator changes at each consensus execution.

A similar analysis gives an average latency of $\frac{5D}{3}$ for *abroadcasting* a message using the Mostéfaoui-Raynal consensus algorithm within the Chandra-Toueg atomic broadcast and if the consensus coordinator changes at each consensus execution.

B.2.2 *TokenFD* atomic broadcast

The following paragraphs present the analytical latency of the *TokenFD* atomic broadcast algorithm in the two location model. As previously shown in Figure 8.1(b) on page 139, a token circulates among the three processes and the set of messages in the token proposal is *adelivered* by each token holder (which then proposes a new set of undelivered messages). The time needed to pass the token between processes is not uniform : indeed, it is negligible when the token is passed between two processes on the local location whereas it is equal to D when passed between a local and a distant location or vice-versa. Due to this non-uniformity, the set of messages transported by the token is not the same for all tokens, which in turn influences in the latency of atomic broadcast.

Figure B.3 shows the token circulation (top part) and an analysis of

which token contains the messages *abroadcast* by the three processes (bottom part). (Almost) all messages sent by p_2 and p_0 are later contained in token I sent by p_2 : indeed, if a message is *abroadcast* between t_2 and t_4 , it does not reach p_1 before t_3 and is therefore not added to the token III sent by p_1 (but later added to the token I sent by p_2). Similarly, all messages sent by p_1 are ordered in the token III sent by p_1 . Finally, token II, sent by p_0 contains only the messages received by the local hosts in the interval between t_1 and t_2 which is negligible in our model. To summarize, two-thirds of the messages are ordered in token I sent by p_2 , whereas the remaining third is ordered in token III sent by p_1 .

Phase (1): Message diffusion The latency of the message diffusion in the *TokenFD* atomic broadcast algorithm is negligible. Indeed, messages sent by p_2 and p_1 are later contained in tokens sent by p_2 and p_1 respectively, and therefore have no diffusion cost. Messages sent by p_0 are ordered in tokens sent by p_2 , resulting in a negligible diffusion cost, since p_0 and p_2 are both on the local location.

Phase (2): Waiting phase The cost of the waiting phase in the *TokenFD* atomic broadcast algorithm is the following. Messages sent by p_0 and p_2 need to wait until the token is held by p_2 to be ordered. On average, this translates into a waiting time of D . The cost of the waiting phase for the messages sent by p_1 is derived in the same way and also yields an average waiting time of D .

Phase (3): Ordering phase There are only two tokens that contain (almost) all unordered messages. Token I, sent by p_2 , reaches p_0 after a negligible amount of time. Process p_0 then sends updates to p_2 (negligible latency) and p_1 (latency of D) about the ordered messages. The average latency of the ordering phase over all processes of token I is thus $\frac{D}{3}$.

Token III, sent by p_1 reaches p_2 after D time units. Process p_0 receives an update from p_2 shortly after, whereas p_1 receives the update after D additional time units. The average latency of the ordering phase over all processes of token III is thus $\frac{4D}{3}$.

Since $\frac{2}{3}$ of the messages are ordered in token I and $\frac{1}{3}$ in token III, the average ordering latency over all messages is $\frac{2D}{3}$.

Summary By summing the latencies of the three phases, this gives an average latency of $\frac{5D}{3}$ for *abroadcasting* a message using the *TokenFD* algorithm. These results are summarized in Table B.5(c) on page 216.

List of publications

Published parts of this thesis

Chapter 2

- [ES05] R. Ekwall and A. Schiper. Replication: Understanding the Advantage of Atomic Broadcast over Quorum Systems. *Journal of Universal Computer Science*, 11(5):703–711, 2005.

Chapter 4

- [ESU04a] R. Ekwall, A. Schiper, and P. Urbán. Token-based atomic broadcast using unreliable failure detectors. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems (SRDS)*, Florianópolis, Brazil, October 2004.
- [ESU04b] R. Ekwall, A. Schiper, and P. Urbán. Token-based Atomic Broadcast using Unreliable Failure Detectors. Technical Report IC/2004/40, École Polytechnique Fédérale de Lausanne, Switzerland, May 2004.
- [ES03] R. Ekwall and A. Schiper. Revisiting token-based atomic broadcast algorithms. Technical Report IC/2003/39, École Polytechnique Fédérale de Lausanne, Switzerland, February 2003.

Chapter 5

- [ES06b] R. Ekwall and A. Schiper. Solving Atomic Broadcast with Indirect Consensus. In *IEEE International Conference on Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, USA, June 2006.
- [ES06c] R. Ekwall and A. Schiper. Solving Atomic Broadcast with Indirect Consensus. Technical Report LSR-Report-2006-001, École Polytechnique Fédérale de Lausanne, Switzerland, 2006.

Chapter 6

- [EUS03] R. Ekwall, P. Urbán, and A. Schiper. Robust TCP Connections for Fault Tolerant Computing. *Journal of Information Science and Engineering*, 19(3):503–516, 2003.
- [EUS02] R. Ekwall, P. Urbán, and A. Schiper. Robust TCP Connections for Fault Tolerant Computing. In *Proc. 9th International Conference on Parallel and Distributed Systems (ICPADS)*, Chung-li, Taiwan, 2002.

Chapter 8

- [ES07] R. Ekwall and A. Schiper. Modeling and Validating the Performance of Atomic Broadcast Algorithms in High Latency Networks. In *Proceedings of Euro-Par (Euro-Par 2007)*, Rennes, France, August 2007.
- [ES06a] R. Ekwall and A. Schiper. Comparing Atomic Broadcast Algorithms in High Latency Networks. Technical Report LSR-Report-2006-003, École Polytechnique Fédérale de Lausanne, Switzerland, July 2006.

Publications related to group communication

- [RMES07] O. Rützi, S. Mena, R. Ekwall, and A. Schiper. On the Cost of Modularity in Atomic Broadcast. In *IEEE International Conference on Dependable Systems and Networks (DSN 2007)*, Edinburgh, UK, June 2007.
- [KE05a] A. Kupšys and R. Ekwall. Architectural Issues of JMS Compliant Group Communication. In *4th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2005)*, Cambridge, MA, USA, 2005.
- [EMPS04a] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards Flexible Finite-State-Machine-Based Protocol Composition. In *3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA 2004)*, Cambridge, MA, USA, 2004.
- [EPS04] R. Ekwall, S. Pleisch, and A. Schiper. Implementing Group Communication Protocols using SDL. In *Proceedings of the*

International IMS Forum 2004, pages 333–340, Cernobbio, Italy, 2004.

[KE05b] A. Kupšys and R. Ekwall. Architectural Issues of JMS Compliant Group Communication. Technical Report IC/2005/16, École Polytechnique Fédérale de Lausanne, Switzerland, 2005.

[EMPS04b] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards Flexible Finite-State Machine Based Protocol Composition. Technical Report IC/2004/63, École Polytechnique Fédérale de Lausanne, Switzerland, 2004.

List of publications

Curriculum Vitæ

I was born in Jönköping (Sweden) in 1980 and am a Swiss and Swedish national. I attended primary and secondary school in Paris (France), Hanoi (Vietnam) and in Founex, in the vicinity of Geneva (Switzerland). In 1997, I graduated from the *International School of Geneva, La Châtaigneraie* and obtained the Maturité Fédérale (Swiss baccalaureat). I started studying Computer Science at *École Polytechnique Fédérale de Lausanne* (EPFL, Switzerland) in October 1997. Between August 1999 and May 2000, I spent two semesters at *Carnegie Mellon University* in Pittsburgh (United States) as an exchange student and was awarded the School of Computer Science Dean's List twice. I then graduated from EPFL in 2002 with a M.Sc. degree in Computer Science, with three distinctions.

Since May 2002, I have been working at the Distributed Systems Laboratory (LSR, EPFL) as a research and teaching assistant, and as a PhD student, under the guidance of Professor André Schiper. In the context of the work at LSR, I participated in the European IST REMUNE project.