EVALUATING THE PERFORMANCE OF DISTRIBUTED AGREEMENT ALGORITHMS: TOOLS, METHODOLOGY AND CASE STUDIES

THÈSE N^{O} 2824 (2003)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut d'informatique fondamentale

SECTION DES SYSTÈMES DE COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Péter URBÁN

M.Sc. in Technical Informatics, Technical University of Budapest, Hongrie
et de nationalité hongroise

acceptée sur proposition du jury:

Prof. A. Schiper, directeur de thèse
Prof. A. Bondavilli, rapporteur
Prof. J.-Y. Le Boudec, rapporteur
Prof. S. Toueg, rapporteur
Prof. J.-D. Decotignie, rapporteur

Lausanne, EPFL
2003

Abstract

Nowadays, networked computers are present in most aspects of everyday life. Moreover, essential parts of society come to depend on distributed systems formed of networked computers, thus making such systems secure and fault tolerant is a top priority. If the particular fault tolerance requirement is high availability, replication of components is a natural choice. Replication is a difficult problem as the state of the replicas must be kept consistent even if some replicas fail, and because in distributed systems, relying on centralized control or a certain timing behavior is often not feasible.

Replication in distributed systems is often implemented using group communication. Group communication is concerned with providing high-level multipoint communication primitives and the associated tools. Most often, an emphasis is put on tolerating crash failures of processes. At the heart of most communication primitives lies an agreement problem: the members of a group must agree on things like the set of messages to be delivered to the application, the delivery order of messages, or the set of processes that crashed.

A lot of algorithms to solve agreement problems have been proposed and their correctness proven. However, performance aspects of agreement algorithms have been somewhat neglected, for a variety of reasons: the lack of theoretical and practical tools to help performance evaluation, and the lack of well-defined benchmarks for agreement algorithms. Also, most performance studies focus on analyzing failure free runs only. In our view, the limited understanding of performance aspects, in both failure free scenarios and scenarios with failure handling, is an obstacle for adopting agreement protocols in practice, and is part of the explanation why such protocols are not in widespread use in the industry today. The main goal of this thesis is to advance the state of the art in this field.

The thesis has major contributions in three domains: new tools, methodology and performance studies. As for new tools, a simulation and prototyping framework offers a practical tool, and some new complexity metrics a theoretical tool for the performance evaluation of agreement algorithms. As for methodology, the thesis proposes a set of well-defined benchmarks for atomic broadcast algorithms (such algorithms are important as they provide the basis for a number of replication techniques). Finally, three studies are presented that investigate important performance issues with agreement algorithms.

The prototyping and simulation framework simplifies the tedious task of devel-

oping algorithms based on message passing, the communication model that most agreement algorithms are written for. In this framework, the same implementation can be reused for simulations and performance measurements on a real network. This characteristic greatly eases the task of validating simulation results with measurements (or vice versa).

As for theoretical tools, we introduce two complexity metrics that predict performance with more accuracy than the traditional time and message complexity metrics. The key point is that our metrics take account for resource contention, both on the network and the hosts; resource contention is widely recognized as having a major impact on the performance of distributed algorithms. Extensive validation studies have been conducted.

Currently, no widely accepted benchmarks exist for agreement algorithms or group communication toolkits, which makes comparing performance results from different sources difficult. In an attempt to consolidate the situation, we define a number of benchmarks for atomic broadcast. Our benchmarks include well-defined metrics, workloads and failure scenarios (faultloads). The use of the benchmarks is illustrated in two detailed case studies.

Two widespread mechanisms for handling failures are unreliable failure detectors which provide *inconsistent* information about failures, and a group membership service which provides *consistent* information about failures, respectively. We analyze the performance tradeoffs of these two techniques, by comparing the performance of two atomic broadcast algorithms designed for an asynchronous system. Based on our results, we advocate a combined use of the two approaches to failure handling.

In another case study, we compare two consensus algorithms designed for an asynchronous system. The two algorithms differ in how they coordinate the decision process: the one uses a centralized and the other a decentralized communication schema. Our results show that the performance tradeoffs are highly affected by a number of characteristics of the environment, like the availability of multicast and the amount of contention on the hosts versus the amount of contention on the network.

Famous theoretical results state that a lot of important agreement problems are not solvable in the asynchronous system model. In our third case study, we investigate how these results are relevant for implementations of a replicated service, by conducting an experiment in a local area network. We exposed a replicated server to extremely high loads and required that the underlying failure detection service detects crashes very fast; the latter is important as the theoretical results are based on the impossibility of reliable failure detection. We found that our replicated server continued working even with the most extreme settings. We discuss the reasons for the robustness of our replicated server.

Résumé

De nos jours, les réseaux informatiques sont omniprésents dans notre vie quotidienne, et notre société est devenue de plus en plus dépendante de systèmes informatiques répartis. Par conséquent, garantir la sûreté de fonctionnement (p.ex., sécurité, tolérance aux pannes) de tels systèmes est devenu un objectif de première importance. Afin de garantir une certaine tolérance aux pannes et assurer un haut degré de disponibilité d'un système, il est naturel de chercher à en répliquer les composants. Néanmoins, la réplication est difficile à mettre en œuvre dans les systèmes répartis. La principale difficulté consiste à maintenir la cohérence des réplicas, tout en s'affranchissant d'un contrôle centralisé.

La réplication dans les systèmes répartis est souvent mise en œuvre au moyen de mécanismes de communication de groupe. La communication de groupe est constituée de diverses primitives de communication de haut niveau avec destinations multiples, ainsi que d'outils associés. Dans la plupart des cas, l'accent est mis sur la tolérance aux pannes de processus de type *fail-stop*. Au cœur de ces primitives de communication, se trouve généralement un problème d'accord réparti ; par exemple, les membres d'un groupe doivent décider, d'un commun accord, l'ensemble des messages qu'ils vont remettre à l'application, l'ordre de livraison de ces messages, ou bien la liste des processus supposés être tombés en panne.

Beaucoup d'algorithmes permettant de résoudre des problèmes d'accord ont été proposés et prouvés correct. Par contre, l'aspect performance de ces algorithmes a rarement été traité en profondeur. Cela est lié à plusieurs raisons dont les principales sont le manque d'outils, théoriques autant que pratiques, permettant de faciliter l'analyse de performance, ainsi que la quasi absence de bancs d'essais standards. D'autre part, le fait que la plupart des analyses sont restreintes à des exécutions sans défaillances est une raison supplémentaire. Selon nous, le manque de compréhension des aspects performance, tant pour les cas avec défaillances que sans, se pose comme un important facteur limitatif pour l'adoption en pratique de protocoles d'accord tolérants aux pannes. Cela peut notamment expliquer leur faible utilisation dans l'industrie. L'objectif principal de cette thèse est de faire progresser l'état de l'art dans ce domaine.

Cette thèse présente des contributions majeures dans trois domaines : des outils nouveaux, une méthodologie et des études de performance. Sur le plan utilitaire, nous présentons un outil pratique ainsi qu'un outil théorique. L'outil pratique proposé est un environnement de simulation et de prototypage pour évaluer les

performances d'algorithmes répartis. L'outil théorique est un jeu de mesures de complexité pour algorithmes répartis. Sur le plan méthodologique, cette thèse propose un ensemble de bancs d'essai pour algorithmes de diffusion atomique (ces algorithmes sont à la base de nombreuses techniques de réplication). Pour finir, trois études sont présentées, qui examinent divers aspects importants liés à la performance des algorithmes d'accord.

L'environnement de simulation et de prototypage simplifie la tâche ardue du développement d'algorithmes basés sur les échange de messages, qui est le modèle le plus utilisé pour décrire les algorithmes d'accord. Dans cet environnement, une même implémentation peut être utilisée aussi bien pour des simulations que pour des mesures de performance dans un réseau réel. Cette caractéristique est importante car elle facilite la validation des simulations au moyen de mesures, et vice-versa.

Au niveau des outils théoriques, deux métriques de complexité sont introduites. Elles permettent d'estimer la performance réelle d'algorithmes répartis de manière plus précise que les métriques traditionnelles de complexité en temps et en messages. L'intérêt principal de nos métriques est qu'elles prennent en compte la contention des ressources partagés, aussi au niveau du réseau que des processeurs des machines. La contention est considérée comme ayant une influence significative sur les performances des algorithmes répartis. Nos métriques ont été validés de manière extensive.

Actuellement, il n'existe malheureusement pas encore de bancs d'essai standards pour évaluer des algorithmes d'accord ou des systèmes de communication de groupe. Ceci rend périlleuse la comparaison de résultats de performance provenant de sources différentes. Pour remédier à cette situation, nous définissons plusieurs bancs d'essai pour algorithmes de diffusion atomique. Ceux-ci incluent des métriques, des charges de travail et des scénarios avec défaillances clairement définis. L'utilisation des bancs d'essai proposés est illustrée au moyen de deux études de performance détaillées.

Deux mécanismes répandus pour le traitement des défaillances sont les détecteurs de faute, qui fournissent une information *incohérente* sur l'occurrence de pannes de processus, et le service de composition de groupe, qui fournit une information *cohérente*. Nous analysons le rapport entre les performances de ces deux techniques, en menant une comparaison de deux algorithmes de diffusion atomique dans un système asynchrone. Les résultats indiquent qu'une utilisation combinée de ces deux approches offre les meilleures performances.

Dans l'étude suivante, deux algorithmes de consensus sont comparés dans un système asynchrone. La différence principale entre ces algorithmes réside dans l'utilisation d'un schéma de communication centralisé pour l'un et décentralisé pour l'autre. Les résultats obtenus démontrent que leurs performances respectives dépendent de nombreuses caractéristiques liées à l'environnement, tel que la disponibilité d'un mécanisme de diffusion au niveau du réseau ou le degré de contention sur les machines par rapport à celui du réseau.

Des résultats théoriques indiquent que beaucoup d'importants problèmes d'ac-

cord ne peuvent pas être résolus dans un modèle asynchrone. Dans la dernière étude de performance de cette thèse, nous cherchons à comprendre l'importance pratique de ces résultats par le biais d'expériences dans un réseau local. Nous avons soumis un serveur répliqué à des charges très élevées, tout en maintenant un service de détection de pannes efficace ; ce dernier point est essentiel car les résultats théoriques se basent sur l'impossibilité de détecter des pannes avec certitude. Le serveur répliqué fonctionnait, y compris avec les valeurs de paramètres les plus extrêmes. L'étude se termine donc sur une discussion des raisons possibles d'un tel comportement.

To my son Dani, born together with this thesis

Acknowledgments

First of all, I would like to thank my supervisor, Prof. André Schiper for all he thought me about doing academic research, the hard work that he put into each of our papers, and his support in general during the five years I spent in the Distributed Systems Laboratory.

I am also very grateful to Xavier Défago for his helpful collaboration and support throughout my Ph.D. research, especially during the initial period, as well as his friendship. This thesis would have been very different without him.

I would also like to thank the CSEM Swiss Center for Electronics and Microtechnology, Inc., based in Neuchâtel, for the financial support of my Ph.D. research. I am especially grateful to Prof. Jean-Dominique Decotignie for his role in setting up this opportunity.

My thanks go to all other people with whom I had the opportunity to collaborate on papers important for this thesis: Prof. Andrea Bondavalli, David Cavin, Andrea Coccoli, Fernando Pedone, Ilya Shnayderman and Naohiro Hayashibara. I have a fond memory of all our heated discussions. I am also grateful to Paweł Wojciechowski and Sergio Mena for their comments on microprotocol frameworks, Prof. Jean-Yves Le Boudec for his numerous advice in the early stages of the work on contention-aware metrics, Prof. Thomas Liebling and Alain Prodon for the discussion about the tool to evaluate these metrics, and Prof. Danny Dolev and Gregory Chockler for their insightful comments on uniformity and its interplay with group membership.

Reviewing a Ph.D. thesis takes a lot of time and effort. For this, and their interesting ideas, I would like to thank the members of the jury, Prof. Andrea Bondavalli, Prof. Jean-Dominique Decotignie, Prof. Jean-Yves Le Boudec and Prof. Sam Toueg, as well as the president of the jury, Prof. Emre Telatar.

I would also like to express my gratitude to all the people in the Operating System Lab and the Distributed Systems Lab for their support and friendship. In particular, I would like to thank Matthias Wiesmann for always being available to discuss about literally everything, often from a funny point of view, as well as Arnas Kupšys, Roel Vandewall, Richard Ekwall, Yoav Sasson and Stefan Pleisch of all people not mentioned so far. My special thanks go to Xavier, André and Matthias for improving my French (including the abstract of this thesis). I would also like to thank both our secretaries, Kristine Verhamme and France Faille for all their kind help.

Last but not least, I wish to express my gratitude to my teachers, family, and close friends, especially to my parents for their loving support throughout my life, my wife Éva for her cheerfulness among all the wonderful qualities she has, and Prof. András Pataricza for making me interested in scientific research.

Contents

1 Introduction **1**
1.1 Research context and motivation 1
1.2 Research contributions 3
1.3 Roadmap for the thesis 5

2 System models and definitions **7**
2.1 System models 7
2.1.1 Synchrony 7
2.1.2 Failure modes 9
2.1.3 Oracles for the asynchronous model 10
2.2 Agreement problems 12
2.2.1 Consensus 13
2.2.2 Reliable broadcast 14
2.2.3 Atomic broadcast 14
2.2.4 Group membership and view synchronous communication 15
2.2.5 Solving agreement problems in asynchronous systems . . 16

3 Introduction to performance evaluation **19**
3.1 A systematic approach to performance evaluation 19
3.2 Performance metrics 20
3.3 Selecting evaluation techniques 22
3.4 Statistical tools 23
3.4.1 Summarizing data 23
3.4.2 Comparing systems using sample data 24
3.4.3 Multiple linear regression 25
3.5 Running experiments 27
3.5.1 Transient removal 27
3.5.2 Stopping criteria 28
3.5.3 Random number generation 29

4 Neko: A single environment to simulate and prototype distributed algorithms **31**
4.1 Neko feature tour 32

4.1.1 Architecture . . . 32
4.1.2 Sample application: farm of processors . . . 35
4.1.3 Easy startup and configuration . . . 36
4.1.4 Simulation and distributed executions . . . 39
4.2 Networks . . . 41
4.2.1 Real networks . . . 41
4.2.2 Simulated networks . . . 42
4.3 Algorithms . . . 43
4.4 Related work . . . 44

5 Contention-aware performance metrics 47
5.1 Introduction . . . 47
5.2 Related work . . . 49
5.3 Distributed system model . . . 50
5.3.1 Overview of the model . . . 50
5.3.2 Resource conflicts . . . 51
5.3.3 Messages sent to oneself . . . 52
5.3.4 Multicast messages . . . 53
5.3.5 Illustration . . . 53
5.4 Definition of the contention-aware metrics . . . 54
5.4.1 Latency metric . . . 54
5.4.2 Throughput metric . . . 55
5.4.3 Mathematical properties of the metrics . . . 56
5.5 Tool to compute the metrics . . . 56
5.5.1 Building an activity network for an execution . . . 56
5.5.2 Parametric computations . . . 57
5.5.3 Summary: the algorithm . . . 58
5.6 Atomic broadcast algorithms . . . 59
5.6.1 Sequencer . . . 59
5.6.2 Privilege-based . . . 60
5.6.3 Communication history . . . 61
5.6.4 Destinations agreement . . . 62
5.7 Comparison of atomic broadcast algorithms . . . 63
5.7.1 Latency metric . . . 63
5.7.2 Throughput metric . . . 64
5.7.3 Latency and throughput in broadcast networks . . . 66
5.8 Experimental validation . . . 67
5.8.1 Workloads and metrics . . . 67
5.8.2 Computing the metrics . . . 68
5.8.3 Measurements . . . 68
5.8.4 Analyzing the data . . . 69
5.8.5 Results . . . 71

6 Benchmarks for atomic broadcast algorithms 79
6.1 Performance metrics . . . 79
6.1.1 Latency metrics . . . 80
6.1.2 Maximum throughput . . . 81
6.1.3 Notes . . . 82
6.2 Workloads . . . 82
6.3 Faultloads . . . 83
6.3.1 Assumptions about the system . . . 83
6.3.2 Steady state of the system . . . 84
6.3.3 Transient state after a crash . . . 84
6.3.4 Modeling failure detectors . . . 85
6.4 Related work . . . 87

7 Comparison of failure detectors and group membership: performance study of two atomic broadcast algorithms 91
7.1 Algorithms . . . 92
7.1.1 Chandra-Toueg uniform atomic broadcast algorithm . . . 92
7.1.2 Fixed sequencer uniform atomic broadcast algorithm . . . 93
7.1.3 Group membership algorithm . . . 95
7.1.4 Expected performance . . . 96
7.2 Elements of our performance study . . . 96
7.2.1 Performance metrics and workloads . . . 96
7.2.2 Faultloads . . . 97
7.2.3 Modeling the execution environment . . . 97
7.3 Results . . . 97
7.4 Discussion . . . 104

8 Comparing the performance of two consensus algorithms with centralized and decentralized communication schemes 107
8.1 Algorithms . . . 108
8.1.1 The consensus algorithms . . . 108
8.1.2 Optimizations to the consensus algorithms . . . 110
8.1.3 The Chandra-Toueg atomic broadcast algorithm . . . 111
8.2 Elements of our performance study . . . 111
8.2.1 Performance metrics and workloads . . . 111
8.2.2 Faultloads . . . 111
8.2.3 Modeling the execution environment . . . 112
8.3 Results . . . 112
8.3.1 Results in the point-to-point network model . . . 113
8.3.2 Results in the broadcast network model . . . 117
8.4 Discussion . . . 119

9 Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be? 125

9.1 Introduction . . . 126
9.2 Related work . . . 127
9.3 Algorithms . . . 127
9.4 Environment and implementation issues . . . 129
9.4.1 Environment . . . 129
9.4.2 Communication protocols . . . 129
9.4.3 Flow control in the application . . . 130
9.5 How robust is our system? . . . 131
9.5.1 Parameters of the experiment . . . 132
9.5.2 Testing if the atomic broadcast algorithm can deliver messages . . . 132
9.6 Results of our experiment . . . 133
9.7 Discussion . . . 135

10 Conclusion 139

10.1 Research assessment . . . 139
10.2 Open questions and future research directions . . . 141

A Atomic broadcast algorithms 157

A.1 Pseudocode notation . . . 157
A.2 Pseudocode descriptions . . . 158
A.2.1 Non-uniform sequencer algorithm . . . 159
A.2.2 Uniform sequencer algorithm . . . 159
A.2.3 Non-uniform privilege-based algorithm . . . 161
A.2.4 Uniform privilege-based algorithm . . . 162
A.2.5 Uniform communication history algorithm . . . 163
A.2.6 Non-uniform destinations agreement algorithm . . . 164
A.2.7 Uniform destinations agreement algorithm . . . 165
A.3 Formulas for the contention-aware metrics . . . 166
A.3.1 Non-uniform sequencer algorithm (Seq) . . . 166
A.3.2 Uniform privilege-based algorithm (PB) . . . 166
A.3.3 Uniform communication history algorithm (CH) . . . 167
A.3.4 Non-uniform destinations agreement algorithm (DA) . . . 168

B Consensus algorithms 169

B.1 An efficient reliable broadcast algorithm . . . 169
B.2 The Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm . . . 170
B.3 The Mostéfaoui-Raynal $\diamond\mathcal{S}$ consensus algorithm . . . 171

List of Figures

2.1 Possible implementation of a failure detector. 11
2.2 Role of a group communication module. 13

4.1 Architecture of Neko. 32
4.2 Details of a passive layer. 33
4.3 Details of an active layer. 33
4.4 Sample layer that multiplexes messages from the upper layers. . . 34
4.5 Architecture of a sample Neko application: farm of processors. . . 35
4.6 Code example for an active layer. 36
4.7 Code example for a passive layer. 37
4.8 Code example for an intermediate layer. 38
4.9 Example of a Neko configuration file (line numbers are not part of the file). 39
4.10 Code example for initializing an application. 40

5.1 Decomposition of the end-to-end delay (tu=time unit). 50
5.2 Network access policy (executed by the network task). 52
5.3 Simple algorithm in model $\mathcal{M}_{\mathrm{br}}(3, 0.5)$. 54
5.4 Simple algorithm in model $\mathcal{M}_{\mathrm{pp}}(3, 0.5)$ ($m_{i,j}$ denotes the copy of message m_i sent to process p_j). 54
5.5 Overview of the algorithm that computes a parametric formula for the contention-aware metrics. 58
5.6 Representative for sequencer algorithms. 60
5.7 Representative for privilege-based algorithms. 61
5.8 Representative for communication history algorithms. 61
5.9 Representative for destinations agreement algorithms. 62
5.10 Comparison of atomic broadcast algorithms in the point-to-point model ($\mathcal{A} > \mathcal{A}'$ means $\mathcal{A}$ “better than” $\mathcal{A}'$). 65
5.11 Comparison of atomic broadcast algorithms in the broadcast model ($\mathcal{A} > \mathcal{A}'$ means $\mathcal{A}$ “better than” $\mathcal{A}'$). 65
5.12 Latency as a function of λ in the general case. 70
5.13 Relative error of latency for a variety of metrics (1). 73
5.14 Relative error of latency for a variety of metrics (2). 74

5.15 Relative error of latency for the contention-aware metric in model $\mathcal{M}_{\rm pp}(n, \lambda)$ (1). 75
5.16 Relative error of latency for the contention-aware metric in model $\mathcal{M}_{\rm pp}(n, \lambda)$ (2). 76
5.17 Cumulative distribution of the end-to-end transmission time in a lightly loaded Ethernet network. 77

6.1 Quality of service metric related to the completeness of failure detectors. Process q monitors process p. 86
6.2 Quality of service metrics related to the accuracy of failure detectors. Process q monitors process p. 86

7.1 Example run of the atomic broadcast algorithms. Labels on the top/bottom refer to the FD/GM algorithm, respectively. 94
7.2 Latency vs. throughput with the normal-steady faultload. 98
7.3 Latency vs. throughput with the crash-steady faultload. In each graph, the legend lists the curves from the top to the bottom. . . . 99
7.4 Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 0.1$). 101
7.5 Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 1$). 101
7.6 Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 10$). 102
7.7 Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 0.1$). 102
7.8 Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 1$). 103
7.9 Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 10$). 103
7.10 Latency overhead vs. throughput with the crash-transient faultload. 105

8.1 Example run of the CT consensus algorithm. 109
8.2 Example run of the MR consensus algorithm. 109
8.3 Latency vs. number of processes with the normal-steady faultload (point-to-point model). 113
8.4 Latency vs. throughput with the normal-steady faultload ($\lambda = 0.1$, point-to-point model). 114
8.5 Latency vs. throughput with the normal-steady faultload ($\lambda = 1$, point-to-point model). 115
8.6 Latency vs. throughput with the normal-steady faultload ($\lambda = 10$, point-to-point model). 115
8.7 Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 0.1$, point-to-point model). 117

8.8 Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 1$, point-to-point model). 118
8.9 Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 10$, point-to-point model). 118
8.10 Latency vs. number of processes with the normal-steady faultload (broadcast model). 119
8.11 Latency vs. throughput with the normal-steady faultload ($\lambda = 0.1$, broadcast model). 120
8.12 Latency vs. throughput with the normal-steady faultload ($\lambda = 1$, broadcast model). 121
8.13 Latency vs. throughput with the normal-steady faultload ($\lambda = 10$, broadcast model). 121
8.14 Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 0.1$, broadcast model). 122
8.15 Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 1$, broadcast model). 123
8.16 Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 10$, broadcast model). 123

9.1 Heartbeat failure detection. 129
9.2 Performance of the replicated server (three replicas) for an extreme and a moderate request rate r vs. the failure detection timeout T. Each point represents a mean value obtained from 100 independent experiments. The 95% confidence interval is shown. 134
9.3 The distribution of the number of rounds per consensus execution, for $r = 20\,000/s$, $T = 1$ ms. 621 consensus executions are shown, coming from 10 independent experiments. The distribution of small round numbers is shown separately as well. 135
9.4 Consensus algorithm: q must suspect p before the reception of m_3 to prevent a decision in the current round. The late process is not shown. 137

List of Tables

3.1 Criteria for selecting an evaluation technique [Jai91]. 22

4.1 Performance comparison of Neko and Java sockets. 42

5.1 Latency metric: evaluation of atomic broadcast algorithms (in model $\mathcal{M}_{\text{pp}}(n, \lambda)$). 64
5.2 Throughput metric: evaluation of atomic broadcast algorithms (in model $\mathcal{M}_{\text{pp}}(n, \lambda)$). 66
5.3 $\text{Latency}_{\text{br}}(\mathcal{A})(n, \lambda)$: evaluation of atomic broadcast algorithms. . 66
5.4 $\text{Thput}_{\text{br}}(\mathcal{A})(n, \lambda)$: evaluation of atomic broadcast algorithms. . . 66
5.5 Estimating the latency of algorithms using a variety of metrics using regression. s_e denotes the standard deviation of errors; the lower, the better. 71

Chapter 1

Introduction

1.1 Research context and motivation

Developments in computing. Nowadays, computers are present in most aspects of everyday life. It is hard to imagine a company without servers and desktop computers, and most people own desktop computers, personal digital assistants, intelligent mobile phones or embedded computers, say in the stereo or the family car. Also, the processing abilities of individual computers have grown tremendously. Another tendency is that these devices are networked more and more: the last two decades have seen the emergence of local and wide area networks, the global Internet, and increasingly ubiquitous wireless networking.

To summarize these developments in one phrase: computers have become much more useful than in the past. This leads to two trends. Firstly, computers are used for more complex tasks than ever. More complex tasks demand more complex systems, be it on the hardware, software or networking (middleware) level. The second trend is that essential parts of society come to depend on computers: banking, telecommunication, and the power grid, just to name a few examples.

The problem of dependability. The two trends clash when it comes to dependability, with all its different aspects like fault tolerance or security. On the one hand, complex systems tend to be more fragile, with frequent failures, as the number of components and their interconnections grow. On the other hand, the more we rely on these systems, the more serious are the consequences of failures. Hardware crashes, software bugs and other problems often have a high cost in terms of money or human lives [Neu03]. For this reason, research on fault tolerance and security is becoming more important.

Fault tolerance can be achieved in a number of ways. The list includes reliable and self-checking hardware components, programming languages, as well as new software engineering and project management methods. If the particular fault tolerance requirement is high availability, replication of components is often a natural choice.

Replication in distributed systems. The basic idea behind replication is simple: several replicas provide the same service, to make sure that if one replica fails, the remaining ones can take over. In tightly coupled systems, replicating components is relatively easy, as means are available to coordinate the operation of the replicas globally; in fact, hardware replication (master-slave configurations, n-modular redundancy) has been a commercially successful field. However, distributed systems are loosely coupled, consisting of relatively complex and autonomous components. In distributed systems, propagating information, or simply synchronizing some components, takes time and may be interleaved with other events in the system. Moreover, the timing behavior is often unpredictable. In a system with such characteristics, special care must be taken to ensure that the state of the replicas remains consistent. Another source of difficulties is that the use of the replicated service is often required to be transparent for the user. Due to these difficulties, replication in distributed systems has been most successful in applications with relaxed requirements: loose consistency (Domain Name Service), stateless operation (serving static web pages) or no full transparency (web transactions in progress may be dropped).

Group communication for replicating services. The field of group communication is concerned with providing high-level multipoint communication primitives and the associated tools. Most often, an emphasis is put on tolerating crash failures of processes, and thus these primitives are especially useful for building replicated services. At the heart of most communication primitives lies an *agreement problem*: the members of a group must agree on things like the set of messages to be delivered to the application, the delivery order of messages, or the set of processes that crashed.

Group communication has been an active research field with both theoretical studies and prototypes. Theoretical results include precisely defined semantics for communication primitives and minimal conditions needed to solve agreement problems (e.g., [FLP85, HT93]). New algorithms from the theoretical community often come with formal proofs of correctness (e.g., [CT96]).

The first prototypes appeared in distributed operating systems [CZ85, KT91a]. The Isis system [BvR94] was the first proof of concept for the toolkit approach, followed by many other toolkits [SDP91, MFSW95, VRBM96, DM96, MMSA$^+$96, MDB01]. Recent research also includes integrating group communication with the middleware infrastructure (e.g., [FGS97]) and studying modular ways of constructing group communication systems [MPR01, VBH$^+$98, SH99, WMS02]. This line of research is mostly concerned with convenient programming abstractions, proof-of-concept systems, and efficient implementation techniques.

Performance of agreement algorithms. Performance evaluation is a central part of every scientific and engineering activity, including the construction of distributed applications. Nevertheless, performance aspects of agreement algorithms have

been somewhat neglected, for a variety of reasons. One reason is the lack of theoretical and practical tools that leads to a gap between the techniques used by engineers and algorithm designers. Engineers of distributed systems rely heavily on various performance evaluation techniques. However, their results are specific to a given system and its environment. In contrast, algorithm designers invest considerable effort into describing algorithms and correctness proofs, but they often oversee the importance of predicting the performance of their algorithms, or just use simple complexity metrics which give highly inaccurate results.

Another reason why performance aspects are often neglected is that there are no widely accepted benchmarks for agreement algorithms, and thus results from different researchers are not comparable. Yet another problem, specific to fault tolerant agreement algorithms, is that most performance studies focus on analyzing failure free runs, thus neglecting the performance aspects of failure handling. This often leads to a skewed view of performance issues, and systems in which the performance of failure handling is sacrificed for marginal gains in failure free system performance.

In our view, the limited understanding of performance aspects, in both failure free scenarios and scenarios with failure handling, is an obstacle for adopting agreement protocols in practice, and is part of the explanation why such protocols are not in widespread use in the industry today. The main goal of this thesis is to advance the state of the art in this field.

1.2 Research contributions

The Neko prototyping and simulation framework. Neko is a distributed programming framework that simplifies the tedious task of writing algorithms based on message passing (most agreement algorithms are described in this communication model). The most important characteristic is that the same implementation can be reused for (1) simulations and (2) performance measurements on a real network. This characteristic makes conducting performance studies easier; in particular, validating simulation results with measurements or vice versa does not require the reimplementation of the algorithm and the experiments. Neko also offers support for centralized configuration, the control of experiments and gathering statistics. A variety of agreement algorithms have been implemented.

Contention-aware performance metrics. To narrow the gap between the techniques used by algorithm designers and engineers, complexity metrics are needed that predict performance with more accuracy than the traditional time and message complexity metrics. We introduce two metrics that, unlike time and message complexity, take account for resource contention, both on the network and the hosts. Resource contention is widely recognized as having a major impact on the performance of distributed algorithms. Beside the metrics themselves, a tool that helps evaluating them is presented, and extensive validation studies are conducted. The

use of the metrics is illustrated in two detailed case studies.

Benchmarks for atomic broadcast algorithms. Currently, no widely accepted benchmarks exist for agreement problems or group communication toolkits, which makes comparing performance results from different sources difficult. The assumptions, metrics, workloads and other issues in most performance studies are wildly different, and are often not described clearly. In an attempt to consolidate the situation, we define a number of benchmarks for atomic broadcast, an important agreement problem requiring that all destinations of messages deliver the same set of messages in the same order.

Our benchmarks include well-defined metrics, workloads and failure scenarios (faultloads). The emphasis is on (1) unambiguous definitions, (2) stating metrics and workloads in terms of send and delivery events, without referring to implementation details of particular atomic broadcast algorithms or fault tolerance mechanisms, (3) simplicity, and (4) extensive treatment of failure scenarios, including not only failures but possibly wrong failure suspicions. The use of the benchmarks is illustrated in two detailed case studies.

Comparing unreliable failure detectors and group membership. The first of our detailed case studies compares the performance of two atomic broadcast algorithms designed for an asynchronous system, i.e., a system with no bounds on processing speeds and message delays. The two algorithms have similar runs if neither failures nor failure suspicions occur. This allows us to concentrate on their differences in how they handle failures. The two algorithms use two widespread mechanisms for handling failures: unreliable failure detectors which provide *inconsistent* information about failures, and a group membership service which provides *consistent* information about failures, respectively. To our knowledge, no quantitative comparison of these two mechanisms has been published so far.

Our results show that constructing consistent failure information for a group membership service has a high cost. On the other hand, it offers long term resiliency and performance benefits, and has a number of uses beside failure handling. Based on these results, we advocate a combined use of the two approaches to failure handling.

Comparing centralized and decentralized coordination. The second of our detailed case studies compares two consensus algorithms designed for an asynchronous system. In the consensus problem, a number of processes propose values, and all participating processes eventually decide the same value which is one of the proposals. The two algorithms make very similar assumptions and exhibit the same basic structure, which makes them ideal candidates for our comparison. They differ in how they coordinate the decision process: the one uses a centralized and the other a decentralized communication schema. The system under test is a

system in which processes send atomic broadcasts; the atomic broadcast algorithm calls the consensus algorithms to decide the delivery order of messages.

Our results show that the performance tradeoffs are highly affected by a number of characteristics of the environment, like the availability of multicast and the amount of contention on the hosts versus the amount of contention on the network. They give hints which algorithm is a more appropriate choice for deployment in a given environment.

On the robustness of replicated servers. A famous theoretical result [FLP85] states that consensus is not solvable in the asynchronous system model. Similar impossibility results have been established for atomic broadcast and group membership, and are thus relevant for implementations of a replicated service. However, the practical impact of these impossibility results is unclear. A question that arises if they set limits to the robustness of a replicated server exposed to extremely high loads.

We investigated the problem by implementing a replicated service and exposed it to high loads in a local area network. As the impossibility results are based on the difficulty of distinguishing slow and crashed processes, we put a focus on the failure detectors underlying the replicated service. In particular, we could control the detection time offered by the failure detectors by tuning a timeout parameter. We found that our replicated server continued working even when the network was overloaded with messages and the detection time for failures was very low (1 ms). Our conclusion is that a replicated service implemented using agreement protocols based on a certain class of failure detectors ($\diamond\mathcal{S}$ [CT96]) is extremely robust under high loads, and that this robustness is maintained even if the service is required to offer a fast reaction to failures. In addition, we discuss a number of implementation decisions that are essential for achieving robustness.

1.3 Roadmap for the thesis

Preliminaries. Chapter 2 introduces models for distributed systems and their failures, and presents formal definitions for the agreement problems that this thesis is concerned with. Chapter 3 gives a short overview of performance evaluation techniques and the statistical tools they rely on.

Tools. Chapter 4 presents the Neko prototyping and simulation framework. All performance studies in this thesis were conducted in this framework. Chapter 5 introduces complexity metrics that take into account resource contention. The metrics are validated with analytical studies and measurements. They are used in all the simulations in this thesis.

Benchmarks. Chapter 6 defines benchmarks for atomic broadcast algorithms that include performance metrics, workloads, as well as faultloads to character-

ize the occurrence of failure related events. The benchmarks are used in the case studies of the two following chapters.

Performance studies. Chapter 7 compares two atomic broadcast algorithms with different failure handling mechanisms: unreliable failure detectors and group membership. Chapter 8 compares two consensus algorithms, one with a centralized pattern of communication and one with a decentralized pattern of communication. Both studies are simulation studies. In contrast, Chapter 9 presents a study with measurements on a real system. A replicated server, using an atomic broadcast algorithm at its core, is put under extremely high load. The mistakes made by the failure detection mechanism are studied in detail; the main issue under study is if frequent wrong suspicions prevent the server from operating, as is predicted by an important theoretical result [FLP85].

Conclusion. Finally, Chapter 10 recalls the main contributions of this work, and discusses some future research directions.

Chapter 2

System models and definitions

2.1 System models

Distributed systems are modeled as a set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ that interact by exchanging messages over communication channels. In a computer network, processes usually correspond to the hosts and channels form the interconnecting network.

There exist a number of models that restrict the behavior of these components, to a greater or smaller extent, and might include other useful components, e.g., clocks. Models have two uses:

Abstract models Distributed algorithms are *designed for* a model of the environment. In other words, they are only guaranteed to work in a certain environment if the assumptions of the model hold in that environment. It is desirable that such models be (1) general, to allow algorithms designed for the model to work in a variety of environments, and (2) simple, to allow a simple expression for algorithms, and easy correctness proofs.

Descriptive models Models can provide a description of existing environments. Their main use is analyzing the performance of distributed algorithms or larger systems. Compared to abstract models, this kind of model is usually detailed and complex, to allow accurate performance estimates, though the level of detail and complexity depends on the goals of the performance study.

In this section, we introduce a number of abstract models. A descriptive model is given later, in Chapter 5.

2.1.1 Synchrony

The synchrony of a model consists of timing assumptions on the behavior of processes and channels. More specifically, one usually considers two main parameters: (1) the relative speed of processes, i.e., the speed ratio of the fastest and the slowest process in the system, and (2) the message transmission delay, i.e., the time that

elapses between the emission and the reception of messages. The synchrony is defined in terms of bounds on these two parameters. The two extremes of synchrony are the following:

Asynchronous model There are no bounds on either the relative speed of processes or the message transmission delay. In fact, there are no assumptions.

Synchronous model There are known upper bounds on both the relative speed of processes and the message transmission delay.

A number of models exist between these two extremes: e.g., *partially synchronous models* [DDS87, DLS88] and the *timed asynchronous model* [CF99]. They relax the assumptions of the synchronous models by considering (1) unknown bounds, (2) bounds that eventually hold, (3) probabilistic bounds, or combinations of these. Other system models hide such assumptions inside a separate module, called an *oracle*. Oracles are discussed in Section 2.1.3.

The asynchronous model is the most general model: algorithms designed for this model work in any environment. However, a lot of interesting problems cannot be solved in this model, hence the interest for less general but more powerful models.[1] *Extended asynchronous models* offer a good compromise between generality and power. Examples include the timed asynchronous model [CF99] or models with failure detector oracles [CT96]. In these models, just as in the asynchronous model, a process can never be sure how long transmitting a given message is going to take, as there are either no known bounds on the communication delay or they are probabilistic. This has two advantages:

- Regardless of the environment, algorithms designed for an extended asynchronous model are *safe*: they never take actions forbidden by the specification.

- Extended asynchronous models describe the most widespread types of computer networks (both local and wide area networks) rather well. In these networks, both the relative speed of processes and the communication delay vary a lot and can reach extreme values.[2] Consider an alternative model, one that sets fixed bounds on these parameters. Such a model is a poor description for common networks, as there is no single good setting for the bounds. On the one hand, if the bounds are set low, the assumptions of the model are often violated. On the other hand, if the bounds are set very high, they are overly pessimistic, and thus algorithms designed for the model may yield poor performance.

All the algorithms in this thesis are designed for asynchronous models: ones with failure detector oracles (see Section 2.1.3).

[1] Beside this reason, a model can be attractive if it offers convenient programming abstractions, e.g., synchronous rounds in the synchronous model.

[2] Real-time systems are a noteworthy exception.

2.1.2 Failure modes

The components of a distributed system, both processes and channels, may fail. It is important to specify the assumptions of the system model on failures, including the expected semantics of failures. This is the subject of this section.

2.1.2.A Process failures

All the algorithms in this thesis assume that processes only fail by *crashing*. A crashed process ceases functioning forever. This means that it stops performing any activity including sending, transmitting and receiving messages. Models including process crashes are also referred to as *crash-stop* or *fail-stop* models.

We call processes that never crash *correct*, and processes that crash *incorrect*. Note that correct/incorrect are predicates over a whole execution: a process that crashes is incorrect even before the crash occurs. Of course, a process cannot determine if it is incorrect (i.e., if it will crash) but these terms are useful, nevertheless, when writing specifications.

In real systems, processes may fail in an arbitrary manner: among other things, they might generate new, unanticipated messages, or change messages (such arbitrary process failures are called Byzantine failures). Nevertheless, we argue that the fail-stop model is realistic, if one assumes that the system is not exposed to malicious attacks: most components of a system use mechanisms that transform Byzantine failures into more benign crash failures. Examples include error detecting codes in memory and processor chips, hardware self-tests, hardware watchdogs, memory protection in operating systems, exception mechanisms in the processor (e.g., division by zero) and in programming languages (e.g., C++ or Java), and certain programming styles (using assertions extensively).

In real systems, processes often recover from crashes, and so called crash-recovery models exist [ACT98] that take recoveries into account. They often require additional components, e.g., stable storage, memory that survives process crashes. Investigating issues related to recovery is out of the scope of this thesis, mainly because modeling the performance in the presence of recoveries would introduce a lot of new parameters. Therefore we stick to the simpler crash-no-recovery model.

Every algorithm needs to assume a bound on the number of process crashes; obviously, if all processes crash, the algorithm cannot proceed. This assumption is part of the system model. We will return to this issue in Section 2.2.5.

2.1.2.B Channel failures

All the algorithms in this thesis assume that channels are *quasi-reliable* [ACT99]. Quasi-reliable channels are defined as follows:

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 process p sends m to another process q and p and q are correct then q eventually receives m.[3]

Quasi-reliable channels are easily implemented over unreliable channels, using error detecting or correcting codes, sequence numbers and retransmission in case of message loss. The TCP protocol is a good approximation of quasi-reliable channels. A protocol that implements the specification more closely is described in [EUS03].

In the literature, one can find the following alternatives to quasi-reliable channels: (1) lossy channels, (2) systems in which processes are not fully connected and (3) systems where partitions can form. (1) and (2) offer a low-level view of the system to the application: message losses are usually masked by the transport layer of the OSI reference model and full connectivity is assured by the network layer. For our purposes, we prefer the more abstract view offered by quasi-reliable channels, for two reasons: it is simpler and there are practical systems (small LANs) where neither message losses nor routing/bridging affect the performance of protocols significantly. As for partitions (3), it is problematic to define them in an (extended) asynchronous model where messages can be subject to very long delays. It is even more problematic to define meaningful metrics for our performance studies in scenarios with partitions: our algorithms either finish quickly in the largest connected set of processes or block for a long time.

2.1.3 Oracles for the asynchronous model

A lot of interesting distributed problems cannot be solved in the asynchronous model. Yet, these problems become solvable if the model is extended with an oracle. An oracle is a component of the distributed system that processes can query in order to get some information that an algorithm can use for guiding its choices. In this section, we introduce failure detectors and other types of oracles.

2.1.3.A Unreliable failure detectors

The notion of unreliable failure detectors was formalized by Chandra and Toueg [CT96] in the context of crash failures. A failure detector can be seen as a set of modules, with one module FD_i attached to every process p_i. A process p_i can query its failure detector module FD_i about the state of other processes; the state can be crashed or not crashed. The information returned by a failure detector can be incorrect: a process not crashed may be suspected to have crashed, and a crashed process may not be suspected. Also, failure detectors may give inconsistent

[3] An alternative specification is *reliable channels* [BCBT96] which require that even messages sent by incorrect processes are eventually received. This requirement is not realistic: real systems typically use buffers and pipeline processing, and thus crashes may affect messages already sent.

information: at a given time t, it is possible that FD_i suspects a process p and FD_j does not suspect p.

Failure detectors are described in terms of a completeness and an accuracy property. The completeness property describes the ability of the failure detector to detect crashes, and the accuracy property restricts the way a failure detector can incorrectly suspect processes which are not crashed. We introduce the following properties:

Strong Completeness Every incorrect process is eventually suspected by every correct process.

Eventual Strong Accuracy There is a time after which no correct process is ever suspected by any correct process.

Eventual Weak Accuracy There is a time after which some correct process is never suspected by any correct process.

We now define two failure detectors: the $\diamond\mathcal{P}$ failure detector satisfies Strong Completeness and Eventual Strong Accuracy, and the $\diamond\mathcal{S}$ failure detector satisfies Strong Completeness and Eventual Weak Accuracy. The asynchronous model with one of these failure detectors is strong enough to solve all the distributed problems we consider in this thesis.

Implementability of failure detectors. We also argue that the asynchronous models with $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ failure detectors are rather general: there are implementations of failure detectors that satisfy the corresponding properties in real systems.

In order to fulfill Strong Completeness, one has to make sure that monitored processes send messages continually; then one can detect, using a simple timeout mechanism, when this stream of messages stops. Figure 2.1 illustrates a possible implementation that uses this technique.

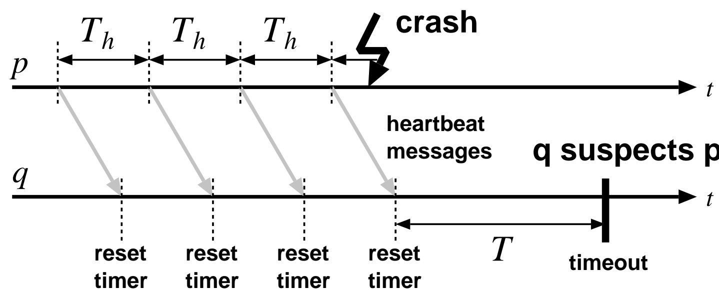


Figure 2.1: Possible implementation of a failure detector.

Ensuring the accuracy properties seems more difficult, because they state that some or all failure detector modules should *never* make mistakes after some point in time. However, note that we only actually need this for a limited duration:

until the algorithm that uses the failure detector finishes or makes some progress. Hence it is enough to make sure that failure detectors do not make mistakes too frequently. This can be achieved by setting the timeout sufficiently high, or setting the timeout adaptively (e.g., increasing the timeout whenever the failure detector makes a mistake).

2.1.3.B Other types of oracles

For completeness, we list other types of oracles for asynchronous systems.

Physical clocks. These oracles return an estimate of the current physical time. One usually assumes that values returned by the same clock increase monotonously, and that clocks on different processes have a bounded difference and a bounded drift. Note that physical clocks by themselves, without additional assumptions, do not help to solve consensus and atomic broadcast.

Coin tosses. Consensus and atomic broadcast can be solved (with probability 1) if algorithms have access to a random number generator (see e.g., [CD89]).

Leader oracles. Such oracles (denoted Ω in [CHT96]) return one trusted process, and it is guaranteed that eventually all oracles on correct processes return the same correct process. $\diamond\mathcal{S}$ and Ω are equally powerful oracles [CT96, CHT96].

Weak ordering oracles. Consensus and atomic broadcast can be solved with this type of oracle [PSUC02a]. This oracle is used to broadcast batches of messages; one batch typically has messages sent by several processes. The oracle offers guarantees of *spontaneous ordering*: sometimes, all processes that receive a batch will receive the *same* message m as the first message of the batch.

2.2 Agreement problems

Agreement problems are a fundamental class of problems in distributed systems. They all follow a common pattern: all participating processes must agree on some common decision, the nature of which depends on the specific problem; e.g., the decision might be the delivery order of messages or the outcome (commit or abort) of a distributed transaction.

Agreement problems are numerous in the context of *group communication*; in fact, this thesis focuses on agreement problems from this field. Group communication is a means for providing point-to-multipoint and multipoint-to-multipoint communication, by organizing processes in groups and defining communication primitives that work with groups. These primitives have richer semantics than the usual point-to-point primitives, in terms of flexibility, ordering guarantees and guarantees of fault tolerance. Ultimately, they ease the construction of certain types

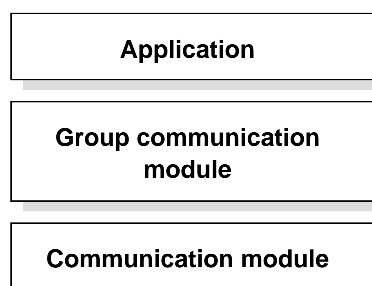


Figure 2.2: Role of a group communication module.

of distributed applications, e.g., fault-tolerant distributed applications. Figure 2.2 sketches the role of a group communication module in a system.

In this section, we give informal and formal definitions for four agreement problems: consensus, reliable broadcast, atomic broadcast, as well as group membership and view synchronous communication. In order to simplify the presentation, we assume that there is only one group that includes all processes in the system (except when defining group membership where the group changes over time).

2.2.1 Consensus

Consensus is a core problem among agreement problems: a lot of agreement problems can be reduced to consensus [GS01], i.e., algorithms that solve these problems can be built using a consensus algorithm. Informally speaking, each participant of a consensus proposes a value, and each of them receives the same decision value, which is one (any one) of the proposed values. More formally, consensus is defined by two primitives propose(v) and decide(v) where v is some value. Uniform consensus is specified as follows (see e.g., [CT96]):

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

Uniform Agreement No two processes decide differently.

Uniform Integrity Every process decides at most once.

Termination Every correct process eventually decides.

Non-uniform consensus ensures fewer guarantees than uniform consensus regarding incorrect processes. Uniform Agreement is replaced by the following property:

Agreement No two *correct* processes decide differently.

Beside consensus, a lot of group communication problems have a uniform and a non-uniform variant (e.g., reliable broadcast and atomic broadcast, defined below). The non-uniform variant allows incorrect processes to take actions (just before they crash) which might never be taken by correct processes. The application developer must consider the consequences of such actions when deciding which variant to use. He/she should keep in mind that ensuring uniformity often has a cost in terms of performance.

2.2.2 Reliable broadcast

Reliable broadcast ensures that all processes deliver a message broadcast previously, even if process crashes occur.[4] Note that this is only difficult to ensure when the sender of the message crashes *while sending* the message: in this case, it is possible that only a subset of all processes receives the message. More formally, reliable broadcast is defined by two primitives R-broadcast(m) and R-deliver(m) where m is some message. Uniform reliable broadcast is specified as follows: [HT93]

Uniform Validity If a correct process R-broadcasts m then it eventually R-delivers m.

Uniform Integrity For any message m, every process R-delivers m at most once, and only if m was R-broadcast by some process.

Uniform Agreement If a process R-delivers m then all correct processes eventually R-deliver m.

Non-uniform reliable broadcast ensures fewer guarantees regarding incorrect processes. Uniform Agreement is replaced by the following property:

Agreement If a *correct* process R-delivers m then all correct processes eventually R-deliver m.

Note that Uniform Validity and Uniform Integrity are rather natural assumptions that should hold for any broadcast service. It is the (Uniform) Agreement property that makes reliable broadcast.

2.2.3 Atomic broadcast

Atomic broadcast is an extension to reliable broadcast: beside ensuring that all processes receive the messages, it also ensures that processes see the messages *in the same order*. More formally, atomic broadcast is defined by two primitives A-broadcast(m) and A-deliver(m) where m is some message. Uniform atomic broadcast is specified with the properties of uniform reliable broadcast (formulated with A-broadcast and A-deliver) plus the following property [HT93]:

[4]Beware: in other fields (e.g., Internet protocols) the term reliable multicast is often used for protocols that tolerate message losses rather than process crashes.

Uniform Total Order For any two processes p and q, if p A-delivers a message m' after message m then q A-delivers m' only after A-delivering m.[5]

Non-uniform atomic broadcast is specified with the properties of non-uniform reliable broadcast plus the following property: [HT93]

Total Order For any two *correct* processes p and q, if p A-delivers a message m' after message m then q A-delivers m' only after A-delivering m.

2.2.4 Group membership and view synchronous communication

The task of a group membership service is to maintain a list of currently active processes. This list can change with new members joining and old members leaving or crashing. Each process has a *view* of this list, and when this list changes, the membership service reports the change to the process by installing a new view. The membership service strives to install the same view at all correct processes.

Our formal specification is mostly based on [CKV01]. The most important difference is that we only define primary component membership here: partitionable membership is out of scope, for reasons listed in Section 2.1.2.B. [CKV01] also lists a lot of group communication systems.

A view V consists of a unique identifier $V.\mathrm{id}$ and a list of processes $V.\mathrm{members}$. For simplicity, we take view identifiers from the set of natural numbers. Processes are notified of view changes by the primitive $\mathrm{view_change}(V)$: we say that a process p *installs* view V. All actions at p after installing V, up to and including the installation of another view V', are said to occur in V. The group membership service satisfies the following properties:

Self Inclusion If a process p installs view V, then p is a member of V.

Local Monotonicity If a process p installs view V after installing view V', then $V.\mathrm{id}$ is greater than $V'.\mathrm{id}$.

Primary Component Membership For every view V (except the first) there exists a previous view V' such that $V.\mathrm{id} = V'.\mathrm{id} + 1$ and a process p that installs V in V'.

Non-triviality Crashed or leaving processes are eventually removed from some view V and all subsequent views. Correct processes that want to join are eventually included in some view V.

Precise Membership If no joins and leaves occur after a certain point in time, then there is a view V such that every correct process installs V as its last view. V includes all correct processes and only those.

[5] A more usual formulation is: If processes p and q both A-deliver m and m' then p A-delivers m before m' if and only if q A-delivers m before m'. However, this specification allows incorrect processes to A-deliver a subset of the sequence of messages and take actions based on the resulting inconsistent state [HT93].

A group membership service is even more useful if one defines how the sending and delivery of messages interact with the service. A variety of multicast services can be defined, with different ordering guarantees, just like in systems with static membership. Here, we only define view synchronous broadcast. Roughly speaking, it ensures that (1) the messages sent in a view V are delivered in the same view V, and that (2) all correct processes in V deliver the same set of messages in V. More formally, view synchronous broadcast is defined by two primitives VS-broadcast(m) and VS-deliver(m) where m is some message, and the following properties (note the similarities to the properties that define non-uniform reliable broadcast in Section 2.2.2):

Initial View Event Every VS-broadcast and VS-deliver occurs within some view.

Uniform Integrity For any message m, every process VS-delivers m at most once, and only if m was VS-broadcast by some process.

Uniform Validity If a correct process VS-broadcasts m then it eventually VS-delivers m.

Sending View Delivery If a process p VS-delivers message m in view V, then the sender of m VS-broadcasts m in view V.

View Synchrony If processes p and q install the same view V in the same previous view V', then any message VS-delivered by p in V' is also VS-delivered by q in V'.

Agreement Let V be the last view mentioned in Precise Membership. If a process VS-delivers m in V then all processes in V eventually VS-deliver m.

In order to satisfy Sending View Delivery without discarding messages from correct processes, processes must stop sending messages for some time before a new view is installed (the group membership service tells the application when it should stop sending messages). One can avoid this blocking period by replacing Sending View Delivery with a weaker property:

Same View Delivery If processes p and q both VS-deliver the message m, they VS-deliver m in the same view V.

2.2.5 Solving agreement problems in asynchronous systems

We now list some theoretical results which are relevant to the agreement problems introduced in this section.

Reliable broadcast. Reliable broadcast can be solved even in the asynchronous model.

Consensus. There is no solution to the consensus problem in the asynchronous model if one (or more) processes can crash [FLP85]. Nevertheless, consensus can be solved in the asynchronous model with $\diamond\mathcal{S}$ failure detectors [CT96]. Also, this model is minimal in a certain sense: [CHT96] proves that it is not possible to weaken the properties of the $\diamond\mathcal{S}$ failure detector and still solve consensus (in fact, the proof is for an equivalent failure detector). Solving consensus in this model (and also in the model with $\diamond\mathcal{P}$) requires that a majority of processes are correct: $f < n/2$ where f is the number of crashes and n is the number of processes.

Atomic broadcast. Atomic broadcast and consensus are equivalent problems [DDS87, CT96], i.e., if there exists an algorithm that solves the one problem, then it can be transformed to solve the other. Hence all results about consensus apply for atomic broadcast.

Group membership. Group membership cannot be solved in the asynchronous model [CHTCB96]. It can be solved with $\diamond\mathcal{P}$ failure detectors [CKV01] if a majority of processes are correct (and there are runs in which more crashes are tolerated).

Failure detectors. There are implementations that satisfy the properties of $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$ in a lot of real systems: see the (practical) discussion in Section 2.1.3.A.

Chapter 3

Introduction to performance evaluation

Performance evaluation is required at every stage in the life cycle of a computer system, including its design, implementation/manufacturing, sales/purchase, use, upgrade, and so on. In this chapter, we introduce some basics of the performance evaluation of computer systems. The purpose of the chapter is to introduce the terminology and the techniques used in this thesis, rather than providing the complete picture. For a more thorough and detailed introduction, we refer the reader to [Jai91]; in fact, most of this chapter is taken from this book.

3.1 A systematic approach to performance evaluation

Most performance problems are unique: the metrics, workloads, and evaluation techniques cannot be used for the next problem. Nevertheless, there are steps common to all performance evaluation projects that help avoid some common mistakes. These steps are listed in this section. This section serves as a roadmap for the rest of the chapter.

1. *State the goals of the study and define the system boundaries.* Given the same set of hardware and software, the definition of the system may vary depending on the goals of the study. Moreover, the choice of system boundaries affects the performance metrics as well as workloads used to compare systems.

2. *List system services and possible outcomes.* Each system provides a set of services, with a number of possible outcomes, some of which are desirable and some of which are not. A list of services and possible outcomes is useful in selecting the right metrics and workloads.

3. *Select performance metrics.* Criteria to compare the performance are called metrics. In general, the metrics are related to the speed, accuracy and avail-

ability of services. The performance of a network, for example, is measured by the speed (throughput and delay), accuracy (error rate) and availability of the packets sent. Performance metrics are discussed in Section 3.2.

4. *List system and workload parameters.* Parameters are variables that affect performance. System parameters generally do not vary among various installations of the system, whereas workload parameters vary from one installation to the next.

5. *Select factors and their values.* The parameters to be varied in a study are called factors and their values are called levels. The parameters that are expected to have a high impact on the performance should be preferably selected as factors. It is important to limit the number of factors and levels, in order to make the study feasible.

6. *Select evaluation techniques.* The choices are analytical modeling, simulation and measurements. They are further discussed in Section 3.3.

7. *Select the workload.* The workload consists of a list of service requests to the system. It is essential that the workload be representative of the system usage in real life. To produce representative workloads, one needs to measure and characterize the workload on existing systems.

   In the context of this thesis, whether and how failures of components occur are often a part of the workload. This part is sometimes called *faultload* [MKA+01].

8. *Design the experiments.* One needs to decide on a sequence of experiments that offer maximum information with minimal effort.

9. *Run the experiments.* Techniques that determine the length and number of experiments are presented in Section 3.5, just after the statistics tools that they rely on (Section 3.4).

10. *Analyze and interpret the data.* The outcomes of measurements and simulations are random quantities. The statistical techniques to correctly analyze them are presented in Section 3.4. Correct interpretation of the results is a key part of the analyst's art, as interpreted results form the basis of decisions.

11. *Present results.* It is important that the results be presented in a manner that is easily understood. This often means presenting the results in graphical form.

3.2 Performance metrics

For each performance study, a set of performance criteria, called *metrics*, must be chosen. One way to prepare this set is to list the services offered by the system.

For each service request made to the system, there are three categories of possible outcomes: the system may perform the service correctly, incorrectly, or refuse to perform the service. For example, a network gateway may forward a packet correctly, or to the wrong destination, or it may be down, in which case it will not forward it at all.

There are metrics associated with each type of outcome. If the system performs the service correctly, its performance is measured by the time taken to perform the service (*responsiveness* metrics), the rate at which the service is performed (*productivity* metrics), and the resources consumed while performing the service (*utilization* metrics). The common name for these metrics is *speed* metrics. For example, the responsiveness of a network gateway is measured by its response time, the time interval between the arrival of a packet and its successful delivery. Its productivity is measured by its throughput, the number of packets forwarded per unit of time. The utilization gives an indication of the percentage of the time the resources of the gateway is busy forwarding packets for a given load level. The resource with the highest utilization is called the *bottleneck*. Performance optimizations at this resource offer the highest payoff.

If the system performs the service incorrectly, an *error* is said to have occurred. It is helpful to classify errors and to determine the probability of errors and/or the time between errors. Such metrics are called *reliability* metrics. For example, the gateway may lose packets from time to time, and the percentage of packets lost may be of interest.

If the system does not perform the service, it is said to be down, failed, or unavailable. Once, again, it is helpful to classify the failure modes and characterize their occurrence. Such metrics are called *availability* metrics. For example, the mean time to failure (MTTF) of the gateway may be one month, and the failure may be due to memory errors in 40% of all cases.

Given a number of metrics, one should use the following considerations to select a subset: low variability, non-redundancy, and completeness. Low variability helps to reach a given level of statistical confidence with fewer repetitions of experiments. Studying redundant metrics may not provide more insight than just studying one of them. Finally, the set of metrics included in the study should be complete, reflecting all possible outcomes. Otherwise, it is easy to optimize a system for just a few metrics, whereas the metrics left out of the study may reach unacceptable values.

Depending upon the utility function of a performance metric, it can be categorizes into three classes:

- *Higher is better (HB).* System throughput is an example.

- *Lower is better (LB).* Response time is an example.

- *Nominal is best (NB).* Both high and low values are undesirable; a particular value in the middle is considered best. System utilization is an example:

	Criterion	Analytical modeling	Simulation	Measurement
1.	Life-cycle stage of system	Any	Any	Post-prototype
2.	Time required	Small	Medium	Varies
3.	Most important tools	Analysts	Computer languages	Instrumentation
4.	Accuracy	Low	Moderate	Varies
5.	Trade-off evaluation of parameters	Easy	Moderate	Difficult
6.	Cost	Small	Medium	High
7.	Saleability of results	Low	Medium	High

Table 3.1: Criteria for selecting an evaluation technique [Jai91].

very high utilization yields high response times, whereas low utilization is a waste of resources.

3.3 Selecting evaluation techniques

The three techniques for performance evaluation are analytical modeling, simulation and measurement. Analytical modeling computes the performance of a system based on a parameterized model of the system and its environment using sophisticated mathematics, e.g., Markov models or queuing systems. In the simulation approach, one builds an executable model of the system and its environment and then runs this model and observes its characteristics. Finally, in the measurement approach, the real system or its prototype is run and observed in an environment similar to the real execution environment.

There are a number of considerations that help decide the technique to be used. These considerations are listed in Table 3.1. The list is ordered from most to least important. A few remarks are appropriate. First of all, accuracy does not guarantee correctness: even highly accurate results may be misleading or plainly wrong, or can be misinterpreted. Also, the time required and the accuracy given by the measurement technique were characterized varying. The reason is simply that Murphy's law strikes measurements more often that other techniques: due to the influence of a high number of environmental parameters which is difficult to account for, if anything can go wrong, it will.

It is helpful to use two or more techniques simultaneously, to verify and validate the results of each one. In fact, until verified and validated, all evaluation results are suspect. This leads us to the following three rules of validation:

- Do not trust the results of a simulation model until they have been validated by analytical modeling or measurements.

- Do not trust the results of an analytical model until they have been validated by a simulation model or measurements.
- Do not trust the results of a measurement until they have been validated by simulation or analytical modeling (most commonly ignored rule).

3.4 Statistical tools

3.4.1 Summarizing data

Summarizing performance data is one of the most common problems faced by performance analysts. The goal is to understand and present the results of the experiments, which can produce a huge number of observations on a given variable.

Summarizing data by a single number. In the most condensed form, a single number may be presented that gives the key characteristics of the data set. This single number is usually called an *average* of the data. This average should be representative of a major part of the data set.

The most frequently used average is called *sample mean*. Given a sample $\{x_1, x_2, \ldots, x_n\}$ of n observations, the sample mean is defined by

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The mean always exists and is unique. It gives equal weight to each observation and, in this sense, makes full use of the sample. A consequence of this is that the mean is affected a lot by outliers, so special care must be taken to identify and exclude outliers. The mean also has a linearity property: the mean of a sum is a sum of the means.

Of course, there are cases when the mean, or even any single number, is not appropriate, because the data set cannot be meaningfully summarized with one number: the values of the data set may be grouped around two or more values.

Summarizing variability. Given a data set, summarizing it by a single number is rarely enough. It is important to include a statement about its variability in any summary of the data. This is because given two systems with the same mean performance, one would generally prefer one whose performance does not vary much from the mean.

Variability is specified most often using the *sample variance*. The sample variance of a sample $\{x_1, x_2, \ldots, x_n\}$ of n observations is defined by

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Note that the sum of squares is divided by $n-1$ and not n. Often, the square root of the sample variance, the *sample standard deviation* s, or the coefficient of variation (C.O.V.), $s/\bar{x}$, is preferred to the sample variance: s is in the unit of the mean, and the C.O.V. is dimensionless, and thus these quantities are easier to understand. Beware: these quantities are affected by outliers.

3.4.2 Comparing systems using sample data

By doing experiments, we can only collect a finite *sample* of data, collected of an infinite *population* of all possibilities. In most real-world problems, the population characteristics, e.g., the population mean, are unknown, and the goal is to estimate these characteristics. The population characteristics, called *parameters*, are fixed values, whereas the sample characteristics, called *statistics*, are random variables. Hence there is a need to specify how precise our estimate of a parameter is, so that we can take well-founded decisions based on the estimate.

One possibility is to use *confidence intervals*. For a parameter μ, we can state something like

$$P\{c_1 \leq \mu \leq c_2\} = 1 - \alpha$$

that is, the probability of μ falling between c_1 and c_2 is $1-\alpha$. The interval $[c_1, c_2]$ is called the *confidence interval* for μ, α is called the *significance level*, and $1-\alpha$ is called the *confidence level*. The levels are often expressed as percentages.

Confidence interval for the mean. The (two-sided) $100(1-\alpha)\%$ confidence interval for the population mean is given by

$$\bar{x} \pm z_{1-\alpha/2} \cdot s/\sqrt{n}$$

where $z_{1-\alpha/2}$ denotes the $(1-\alpha/2)$-quantile of a unit normal variate. Some frequently used coefficients are $z_{0.975} = 1.645$ for a 90% confidence interval, $z_{0.975} = 1.958$ for a 95% confidence interval, and $z_{0.995} = 2.576$ for a 99% confidence interval. The formula is only valid if $n > 30$ (this is necessary to make sure that the central limit theorem applies; the assumption underlying the formula is that the sample mean is normally distributed).

For small samples ($n \leq 30$) taken from a normally distributed population, we have

$$\bar{x} \pm t_{[1-\alpha/2;n-1]} \cdot s/\sqrt{n}$$

where $t_{[1-\alpha/2;n-1]}$ denotes the $(1-\alpha/2)$-quantile of a t-variate with $n-1$ degrees of freedom (see the tables in [Jai91]).

Comparing two systems. The mean and its confidence interval can be used to compare two systems. If one makes *paired observations*, i.e., there is a one-to-one correspondence between the observations of the two systems involved, then one should estimate the mean of the difference of each pair of observation. If the confidence interval for the mean includes zero, the two systems are not significantly different. If the confidence interval does not include zero, the two systems are significantly different, and the sign of the mean indicates which one is better.

If one makes *unpaired observations*, one can estimate the means of the two systems. If the confidence intervals do not overlap, the systems are significantly different. If they overlap and the one mean is in the confidence interval for the other mean, the systems are not significantly different. Otherwise, more advanced statistics should be used [Jai91].

Determining sample size. The larger the sample, the higher is the associated confidence. However, larger samples require more effort. Hence the analyst's goal is often to find the smallest sample size for a desired confidence. Suppose that we want to determine the mean performance with an accuracy of $\pm r\%$ and a confidence level of $100(1-\alpha)\%$. We do some preliminary experiments to obtain $\bar{x}$ and s, then set the desired interval equal to that obtained with n observations, and solve for n:

$$\bar{x} \pm z_{1-\alpha/2} s/\sqrt{n} = \bar{x}\,(1 \pm r/100)$$

$$n = \left(\frac{100 z_{1-\alpha/2} s)}{r\bar{x}}\right)^2$$

3.4.3 Multiple linear regression

Regression model. A regression model allows one to estimate or predict a random variable as a function of several other variables. The estimated variable is called the *response variable*, and the other variables are called *predictor variables*, *predictors*, or *factors*. We only summarize *multiple linear regression*, the technique in which the data is fit to a linear combination of the predictor variables in such a way that the squares of differences of data points and their estimates is minimal.

The regression model allows one to predict a response variable y as a function of k predictor variables $x_1, x_2, \ldots, x_k$ using a linear model of the following form:

$$y = b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_k x_k + e$$

Here, $b_0, b_1, \ldots, b_k$ are $k+1$ fixed parameters and e is the error term.

Given a sample of n observations

$$(x_{11}, x_{21}, \ldots, x_{k1}, y_1), \ldots, (x_{1n}, x_{2n}, \ldots, x_{kn}, y_n)$$

the model consists of the following equations:

$$\begin{aligned} y_1 &= b_0 + b_1 x_{11} + b_2 x_{21} + \ldots + b_k x_{k1} + e_1 \\ y_2 &= b_0 + b_1 x_{12} + b_2 x_{22} + \ldots + b_k x_{k2} + e_2 \\ &\vdots \\ y_n &= b_0 + b_1 x_{1n} + b_2 x_{2n} + \ldots + b_k x_{kn} + e_n \end{aligned}$$

In vector notation, we have

$$\mathbf{y} = \mathbf{X}\mathbf{b} + \mathbf{e}$$

where $\mathbf{y} = [y_1 \ \ldots \ y_n]^T$, $\mathbf{b} = [b_0 \, b_1 \ \ldots \ b_n]$ and $\mathbf{e} = [e_1 \ \ldots \ e_n]$ are column vectors and $\mathbf{X}$ is a matrix with $\mathbf{X}(i, j+1) = 1$ if $j = 0$ and x_{ij} otherwise. The goal is to find the regression parameters $\mathbf{b}$ that minimize $\mathbf{e}^2 = e_1^2 + e_2^2 + \ldots + e_n^2$.

Estimation of model parameters. The regression parameters are estimated by

$$\mathbf{b} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\left(\mathbf{X}^T y\right)$$

A part of the overall variation (SST) of y is explained by the regression model (SSR) and another part is not (SSE). We know that

$$\text{SST} = \text{SSR} + \text{SSE} = \sum y^2 - n\bar{y}^2 \quad \text{and} \quad \text{SSE} = \mathbf{y}^T\mathbf{y} - \mathbf{b}^T\mathbf{X}^T\mathbf{y}$$

The goodness of the regression is measured by the *coefficient of determination* $R^2 = \text{SSR}/\text{SST}$. The inequality $0 \leq R^2 \leq 1$ always holds, and higher values are better.

The standard deviation of errors is given by

$$s_e = \sqrt{\frac{\text{SSE}}{n-k-1}}$$

It is used in computing standard deviations for the regression parameters:

$$s_{b_j} = s_e\sqrt{C_{jj}}$$

where C_{jj} is the j-th diagonal term of $\mathbf{C} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}$.
Confidence intervals for the regression parameters are computed using the t-variate $t_{[1-\alpha/2;n-k-1]}$ (see Section 3.4.2):

$$b_j \pm t_{[1-\alpha/2;n-k-1]} s_{b_j}/\sqrt{n}$$

Assumptions. The multiple linear regression model is based on a number of assumptions that should be verified before the analysis: (1) errors are independent and identically distributed normal variates with zero mean; (2) errors have the same variance for all values of the predictors; (3) errors are additive; (4) x_i's and y are linearly related; (5) x_i's are non-stochastic and are measured without error.

3.5 Running experiments

A number of issues related to running experiments are discussed in this section. They are of concern when one evaluates the performance using simulation or measurements. The considerations apply to the analytical modeling technique as well if the model is solved by simulation.

3.5.1 Transient removal

Most often, the analyst is only interested in the *steady-state performance* of a system, that is, the performance after the system has reached a stable state. The results of the initial part of the experiment, called *transient state*, should not be included into the computations of the data analysis. The problem of identifying the end of the transient state is called *transient removal*. We now introduce three methods for transient removal based on statistical techniques: the initial data deletion technique, the method of moving averages, and the method of batch means.

The first two methods try to identify a *knee* on a graph listing all observations along the x axis. This graph is usually highly variable, hence the points are replaced by means of subsets of the data, in order to get a smoother graph where the knee is more easily identified. At first, the observations are averaged over several replications of the same experiment, to obtain a *mean trajectory*: given m replications of size n, if x_{ij} denotes the j-th observation in the i-th replication,

$$\bar{x}_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$$

is computed.

Initial data deletion technique. This technique continues as follows: one computes the overall mean $\bar{\bar{x}}$ and the mean after deleting the first l observations $\bar{\bar{x}}_l$

$$\bar{\bar{x}} = \frac{1}{n}\sum_{j=1}^{n} \bar{x}_j \quad \text{and} \quad \bar{\bar{x}}_l = \frac{1}{n-l}\sum_{j=l+1}^{n} \bar{x}_j$$

and plots the relative change in the overall mean

$$\frac{\bar{\bar{x}}_l - \bar{\bar{x}}}{\bar{\bar{x}}}$$

for $l = 1, 2, \ldots$ until the knee is found.

Method of moving averages. The method of moving averages of independent replications computes a trajectory of the moving average of $2k + 1$ successive values:

$$\bar{\bar{x}}_j = \frac{1}{1+2k} \sum_{l=-k}^{k} \bar{x}_{j+l} \quad \text{for} \quad j = k+1, k+2, \ldots, n-k$$

This trajectory is plotted for $k = 1, 2, \ldots$ until the resulting curve is smooth enough to allow one to find the knee.

Method of batch means. This method requires running a very long simulation and later dividing it up into several *batches* of equal size. The method studies the variance of the mean of observations in each batch as a function of the batch size. Consider a long run of N observations and divide it into m batches of size n each, where $m = \lfloor N/n \rfloor$. At first, one computes a batch mean for each batch

$$\bar{x}_i = \frac{1}{m} \sum_{j=1}^{n} x_{in+j}$$

and then computes the variance of batch means Var($\bar{x}$). This quantity is plotted against $n = 2, 3, \ldots$. The resulting curve has an irregular first part, and then the variance decreases. The rightmost local maximum of the resulting curve shows the length of the transient interval.

3.5.2 Stopping criteria

From the discussion in Section 3.4.2, it follows that the length of the simulation or measurement experiment must be chosen appropriately, long enough to allow the confidence interval for the mean of an observed quantity to narrow to a desired width. In Section 3.4.2, we discussed how the confidence intervals can be computed, given n independent data values. The problem here is that the observations within the same replication of the experiment are *not* independent. We introduce two methods to produce independent values which can then be used to compute the confidence interval.

Method of independent replications. In this method, one runs several replications of the same experiment, while ensuring that the replications are fairly independent. For each replication, one removes the initial transient interval and computes the mean of all observations. The resulting means are independent random variables.

Method of batch means. In this method, one runs a long experiment and removes the initial transient period. The remaining observations are split up into batches, and the mean of observations in each batch is computed. If the batches are long enough, the means are independent – assuming that there are no long term correlations in the measured data. This assumption can be verified by computing

the covariance of successive batch means; this quantity is also called autocovariance. This analysis is repeated with increasing values for the batch size until the autocovariance of the batch means is small compared to their variance.

3.5.3 Random number generation

The issue of random number generation is specific to simulation. Usually, one uses *random number generators*, functions that produce streams of seemingly random (*pseudo-random*) numbers x_i, with x_{i+1} computed from x_i and x_i only. The starting value x_0 is called the *seed*. The choice of the seed determines the whole stream.

Nowadays, the standard random number generators, provided with the programming language, are of good enough quality to fulfill all the needs of simulations, i.e., pass all the important statistical tests for randomness. The analyst only needs to know how many streams to use and how to choose the seed values for each stream. Some rules are listed next.

- *Do not subdivide one stream.* If you need to generate values for several independent variables, use an independent stream for each. Otherwise, the variables are likely to have a strong correlation.

- *Use non-overlapping streams.* In particular, do not use the same seed for all streams. If you know that any given stream is used only 10000 times, then the following random numbers from the same stream are good seeds: x_0, x_{10000}, x_{20000}, etc. For a lot of generators, e.g., the linear congruential generator used by Java, it is possible to calculate x_i directly, without generating the whole stream.

- *Reuse seeds in successive replications.* There is no need to reinitialize the streams between replications.

- *Do not use truly random seeds,* e.g., ones computed from the current time or by the operating system. If you do, two problems arise: your experiments become non-repeatable, and it is not possible to guarantee that the streams will not overlap.

Chapter 4

Neko: A single environment to simulate and prototype distributed algorithms

The three basic approaches to performance evaluation are analytical modeling, simulation and measurements. Their respective advantages and limitations were discussed in Section 3.3. It was also mentioned that in order to increase the credibility and the accuracy of the analysis, it is considered good practice to compare the results obtained through at least two different approaches.

In spite of its importance in the context of distributed systems, performance engineering often does not receive the attention that it deserves. Part of the difficulties stems from the fact that one usually has to develop one implementation of the algorithm for measurements, and a different implementation (possibly in a different language) for simulations. In this chapter, we propose a solution to this last problem. We present *Neko*, a simple communication platform that allows us to both simulate a distributed algorithm and execute it on a real network, using the *same implementation* for the algorithm. Using Neko thus ultimately results in lower development time for a given algorithm. Beside this main application, Neko is also a convenient implementation platform which does not incur a major overhead on communications. Neko is written in Java and is thus highly portable. It has been deliberately kept simple, extensible and easy to use.

The rest of the chapter is structured as follows. Section 4.1 describes the most important features of Neko. We intend to illustrate the simplicity of using Neko throughout this section. Section 4.2 presents the various types of real and simulated networks that Neko currently supports. Section 4.3 describes algorithms and other building blocks for applications developed with Neko. Section 4.4 discusses other work that relates to Neko.

4.1 Neko feature tour

We start with an overview of Neko. We first present the architecture of the platform, and the most important components seen by application programmers. We then illustrate the use of Neko with a simple application. We also show how to start up and configure the example, to run either as a simulation or on a real network. Finally, we discuss the differences between simulations and distributed executions.

4.1.1 Architecture

As shown in Figure 4.1, the architecture of Neko consists of two main parts: *application* and *networks* (NekoProcess is explained later).

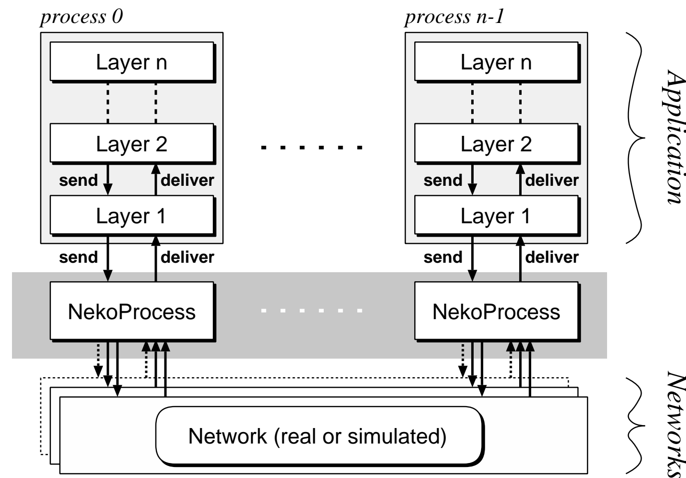


Figure 4.1: Architecture of Neko.

At the level of the application, a collection of *processes* (numbered from 0 to $n-1$) communicate using a simple message passing interface: a sender process pushes its message onto the network with the (asynchronous) primitive send, and the network then pushes that message onto the receiving process with deliver. Processes are programmed as multi-layered programs.

In Neko, the communication platform is not a black box: the communication infrastructure can be controlled in several ways. First, a network can be instantiated from a collection of predefined networks, such as a real network using TCP/IP or simulated Ethernet. Second, Neko can manage several networks in parallel. Third, networks that extend the current framework can easily be programmed and added.

We now present some important aspects of the architecture that are relevant for Neko applications. Details related to the networks are explained in Section 4.2.

Application layers. Neko applications are usually constructed as a *hierarchy of layers*. Messages to be sent are passed down the hierarchy using the method

send, and messages delivered are passed up the hierarchy using the method deliver (Fig. 4.1). Layers are either *active* or *passive*. Passive layers (Fig. 4.2) do not have their own thread of control. Messages are pushed upon passive layers, i.e., the layer below calls their deliver method. Active layers (Fig. 4.3), derived from the class ActiveLayer, have their own thread of control. They actively pull messages from the layer below, using receive (they have an associated FIFO message queue supplied by deliver and read by receive). The call to receive blocks until a message is available. One can also specify a timeout, and a timeout of zero corresponds to a non-blocking call to receive.

Active layers might interact with the layers below using deliver, just like passive layers do (Fig. 4.2). In order to do this, they have to bypass the FIFO message queue of Fig. 4.3 by providing their own deliver method.

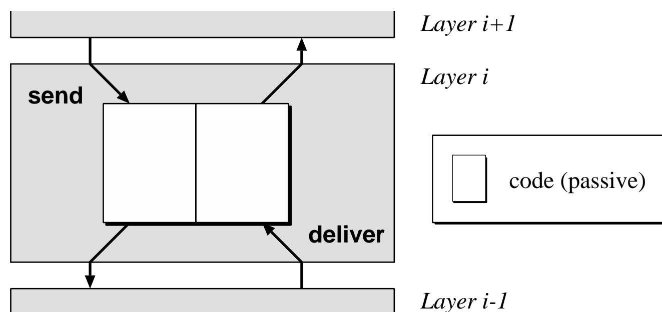


Figure 4.2: Details of a passive layer.

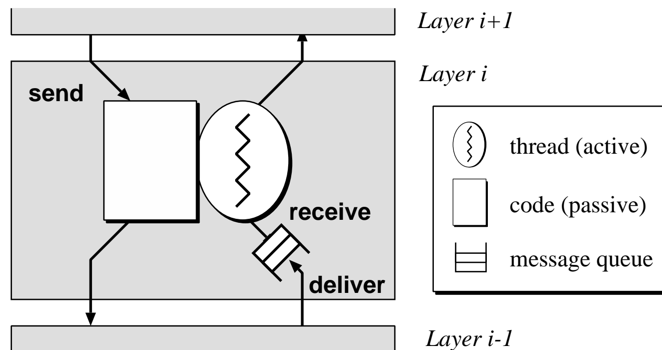


Figure 4.3: Details of an active layer.

Developers are not obliged to structure their applications as a hierarchy of layers. Layers can be combined in other ways: Fig. 4.4 shows a layer which multiplexes messages coming from several layers into one channel (and demultiplexes in the opposite direction), based on the message type. Layers may also interact by calling user-defined methods on each other, i.e., they are not restricted to send and deliver/receive. In general, developers may use Java objects of any type, not just

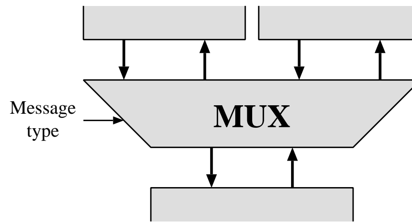


Figure 4.4: Sample layer that multiplexes messages from the upper layers.

layers, arranged and interacting in an arbitrary fashion.

NekoProcess Each process of the distributed application has an associated object of type NekoProcess, placed between the layers of the application and the network (Fig. 4.1). The NekoProcess takes several important roles:

1. It holds some process wide information, e.g., the address of the process. All layers of the process have access to the NekoProcess. A typical use is for Single Program Multiple Data (SPMD) programming: the same program is running on several processes, and it branches on the address of the process on which it is running. This address is obtained from the NekoProcess.

2. It implements some generally useful services, such as logging messages sent and received by the process.

3. If the application uses several networks in parallel (e.g., because it communicates over two different physical networks, or uses two different protocols over the same physical network) the NekoProcess dispatches (and collects) messages to (from) the appropriate network.

NekoMessage All communication primitives (send, deliver and receive) transmit instances of NekoMessages. A message can be either a unicast or a multicast message. Every message is composed of a content part that consists of any Java object, and a header with the following information:

Addressing (source, destinations) The addressing information consists of the address of the sender process and the address of the destination process(es). Addresses are small integers; the numbering of processes starts from 0. This gives a very simple addressing scheme, with no hierarchy.

Network When Neko manages several networks in parallel (see Fig. 4.1), each message carries the identification of the network that should be used for transmission. This can be specified when the message is sent.

Message type Each message has a user-defined type field (integer). It can be used to distinguish messages belonging to different protocols.

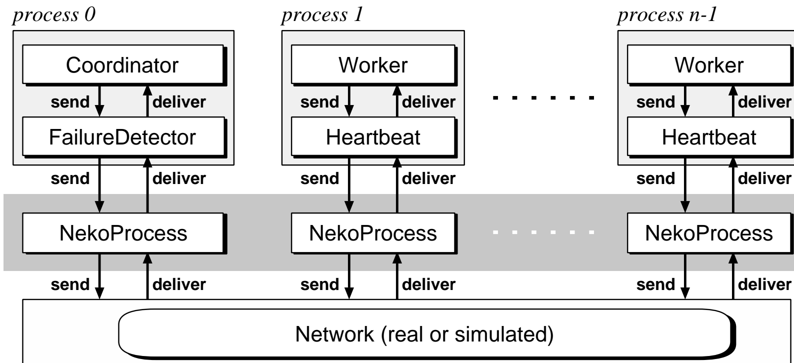


Figure 4.5: Architecture of a sample Neko application: farm of processors.

4.1.2 Sample application: farm of processors

In this section, we illustrate the application layers using an example. The example is explained in detail in order to highlight how easily one can develop distributed applications with Neko. Some more complex applications are described in Section 4.3.

Our example is the following: a complex task is divided into sub-tasks, and each sub-task is assigned to one machine out of a pool of machines. When a machine has finished a sub-task, it sends back the result and gets a new sub-task. We also have a fault tolerance requirement. As we use a large number of machines, it is most likely that a few of them are down from time to time. We do not want to assign sub-tasks to these machines.

The implementation has two layers on every process, as shown in Fig. 4.5. We now describe these layers.

Top layers. The Coordinator distributes the task and collects the result, and the Workers do the actual computation. The Worker code is shown in Fig. 4.6 (we only give code for the simplest layers due to space constraints). It is an active layer: active layers extend ActiveLayer which extends NekoThread, which is used similarly to a Java thread. The thread executes its run method. This method uses receive to get a message from the network. The result is computed and sent to the layer below (stored in the sender variable).[1] After going through the protocol stacks, this message is delivered to the Coordinator.

The Worker layer can also be implemented as a passive layer, extending Layer; this implementation is shown in Fig. 4.7. Whenever the layer is delivered a message, it computes some result and sends the message to the layer below (sender).

Bottom layers. By describing the bottom layers, we intend to show here a code example and a few possible uses of the hierarchy of layers.

[1]The layer above is stored in the receiver variable.

```
public class Worker
   extends ActiveLayer
{
   public void run() {
      while (true) {
         // receive work from coordinator
         NekoMessage m = receive();
         // process work
         Object result = compute(m.getContent());
         // send back results
         sender.send(new NekoMessage(0, RESULT, result));
} } }
```

Figure 4.6: Code example for an active layer.

The Heartbeat and FailureDetector layers cooperate to implement failure detection. The basic scheme is that Heartbeat layers send a *heartbeat* message every second, and the FailureDetector starts suspecting a process p if no heartbeat arrived from p in 2 seconds. Upon suspicion, the FailureDetector delivers a NekoMessage containing a notification to the Coordinator, which can then re-assign the sub-task in progress on the faulty process. This illustrates one possible use of the hierarchy of layers: notifying layers about asynchronous events (in this case, about the failure of a process).

Another use of the hierarchy of layers is that lower layers can observe messages going to and coming from higher layers. We can exploit this to optimize the basic scheme for failure detection: let replies from Workers also act as heartbeats, so that we can reduce the number of heartbeats. In other words, a Heartbeat only needs to send an (explicit) heartbeat if no reply has been sent for 1 second, and the Failure Detector only needs to suspect a process p if no reply *nor* heartbeat is received for 2 seconds. The code for a Heartbeat is shown in Fig. 4.8. It is an intermediate layer, with methods send and deliver. The method deliver simply passes on messages; it uses the data member receiver which points to the layer on top (in this case, the Worker). The method send also passes on messages (and uses the data member sender that points to the NekoProcess below) but additionally, it sets the deadline variable, which indicates when the next heartbeat has to be sent. Heartbeat is an active layer, thus it extends ActiveLayer and has its own thread of control. Its run method takes care of sending a heartbeat message whenever the deadline expires.[2]

4.1.3 Easy startup and configuration

Bootstrapping and configuring a distributed application is far from being trivial. In this section, we explain what support Neko provides for this task.

[2]The careful reader might notice that synchronized blocks are missing. The concurrent execution of Neko and the Heartbeat thread may indeed result in heartbeats sent more often than intended, but this does not compromise the integrity of the application.

```
public class Worker
   extends Layer
{
   public void deliver(NekoMessage m) {
      // process work received from coordinator
      Object result = compute(m.getContent());
      // send back results
      sender.send(new NekoMessage(0, RESULT, result));
}  }
```

Figure 4.7: Code example for a passive layer.

Configuration. All aspects of a Neko application are configured by a single file. The name of this file is the only parameter to be passed to Neko on startup. Neko ensures that each process of the application has the information in the configuration file when the application starts. Possible configuration files for the example of Section 4.1.2 are shown in Fig. 4.9; distributed executions need the file in Fig. 4.9(a) and simulations the file in Fig. 4.9(b). The entries are explained in the rest of this section.

Bootstrapping (Fig. 4.9, lines 1-2). Bootstrapping a Neko application is different for a simulation and a distributed execution; this is why some entries of the configuration files differ.

Launching a simulation is simple: only one Java Virtual Machine (JVM) is needed with the name of the configuration file, and Neko will create and initialize all processes. The number of processes is specified by the entry process.num.

For a distributed execution, one has to launch one JVM per process. The process launched last (called *master*) reads the configuration file and interprets the slave entry which lists the addresses of all the other processes (called *slaves*). Then it builds a *control network* including all processes (by opening TCP connections to each slave), and distributes the configuration information.[3]

It is tedious to launch all the JVMs by hand for each execution of the application. For this reason, Neko provides *slave factories* to launch slaves automatically. The slave factories run as daemons on their hosts. Upon bootstrapping, the *master* contacts each slave factory and instructs it to create a new JVM which will act as a slave. Also, there exists another way of bootstrapping, useful in clusters where a command is provided to launch processes on multiple machines, like the mpirun script in an MPI environment [SOHL+98, GHLL+98]. Here, all JVMs are launched at the same time and establish connections based on the information in the configuration file.

[3]The master-slave distinction only exists during bootstrapping. All processes are equal afterwards.

```
public class Heartbeat
   extends ActiveLayer
{
   double deadline = clock() + PERIOD;

   public void send(NekoMessage m) {
      deadline = clock() + PERIOD;
      // PERIOD is 1 second
      sender.send(m);
      // sender is the layer below: the network
   }

   public void deliver(NekoMessage m) {
      receiver.deliver(m);
      // receiver is the Worker
   }

   public void run() {
      while (true) {
         sleep(deadline−clock());
         send(new NekoMessage(0, HEARTBEAT, null));
}  }  }
```

Figure 4.8: Code example for an intermediate layer.

Initialization (Fig. 4.9, lines 3-7). The networks are initialized next (Fig. 4.9, line 3). The names of the classes implementing the networks are given by the network entry, which is, of course, different for simulations and distributed executions. The components of a real network in different JVMs usually need to exchange information upon initialization; the control network, built in the bootstrapping phase, is used for this purpose.

Then comes the initialization of the application (Fig. 4.9, lines 4-6). Each process has an initializer class, given by the process.*i*.initializer entry for process #*i*. The initializer code of process #1 is shown in Fig. 4.10. It constructs the hierarchy of layers in a bottom-up manner by calling addLayer on the NekoProcess. The code has access to all configuration information; it uses the (application specific) entry heartbeat.interval (Fig. 4.9, line 7) to configure the Heartbeat layer.

Execution. Once all the initialization is finished, all NekoThreads are started and the application begins executing. The application has access to the entries of the configuration file.

Shutdown. Terminating a distributed application is also an issue worth mentioning. There is no general (i.e., application independent) solution to this issue. Neko provides a shutdown function that any process can call and which results in shutting down all processes. Processes may implement more complex termination algorithms that end with calling the shutdown function. The termination algorithm

```
1 process.num = 3
2 slave = host1.our.net,host2.our.net
3 network = lse.neko.networks.TCPNetwork
4 process.0.initializer = lse.neko.alg.CoordinatorInitializer
5 process.1.initializer = lse.neko.alg.WorkerInitializer
6 process.2.initializer = lse.neko.alg.WorkerInitializer
7 heartbeat.interval = 1000
```

(a) for a distributed execution

```
1 process.num = 3
2 # no corresponding line
3 network = lse.neko.networks.MetricNetwork
4 process.0.initializer = lse.neko.alg.CoordinatorInitializer
5 process.1.initializer = lse.neko.alg.WorkerInitializer
6 process.2.initializer = lse.neko.alg.WorkerInitializer
7 heartbeat.interval = 1000
```

(b) for a simulation

Figure 4.9: Example of a Neko configuration file (line numbers are not part of the file).

executed by the shutdown function exchanges messages on the control network (built during the bootstrapping phase).

4.1.4 Simulation and distributed executions

One of the main goals of Neko is to allow the same application to run (1) as a simulation and (2) on top of a real network. These two execution modes are fundamentally different in some respects. This section summarizes the (few) rules to follow if the application is to be run both as a simulation and as a distributed application.

No global variables. The first difference is that all processes of a simulation run in the same Java Virtual Machine (JVM),[4] whereas a real application uses one JVM for each process. For this reason, code written for both execution modes should not rely on any global (static) variables. Global variables have two uses in simulations: they either keep (1) information private to one process, or (2) information global to the whole application. In the first case, the information should be accessed using the NekoProcess object as a key. In the second case, there are two choices. Either the information should be distributed using the network, or (if constant and available at startup) it can appear in the configuration file (see Section 4.1.3).

[4]Simulations are not distributed. They only simulate distribution.

```
public class WorkerInitializer
   implements NekoProcessInitializer
{
   public void init(NekoProcess process, Configurations config) {
      Heartbeat heartbeat = new Heartbeat();
      process.addLayer(heartbeat);
      Worker algorithm = new Worker();
      process.addLayer(algorithm);
      heartbeat.setInterval( config.getInteger("heartbeat.interval") );
} }
```

Figure 4.10: Code example for initializing an application.

Threads. The other difference lies in the threading model. Real applications use the class java.lang.Thread, whereas thread based discrete event simulation packages have their special purpose threads, which maintain simulation time and are scheduled according to this simulation time. The two threading models usually do not have the same interface.[5] Both have operations specific to their application area. For example, SimJava [HM98], one of the simulation packages used in Neko, defines channels between active objects as a convenient way to have threads interact. Even the overlapping part of the interfaces is different, e.g., Java threads are started explicitly with start, while SimJava threads are started implicitly upon the start of the simulation.

Neko hides these differences by introducing NekoThread, which encapsulates the common functionality useful for most applications. All threads of the application have to extend this class or use it with a Runnable object. NekoThreads behave like simplified Java threads: in particular, they support wait, notify and notifyAll as a mechanism to synchronize threads. The differences are the following:

- Threads started during the initialization of the application only begin execution when the whole Neko application is started. This simplifies the initialization of the application to a great extent.
- Time is represented with variables of type double in Neko (and the unit is milliseconds). Therefore sleep and wait with a timeout take double as argument.

The class NekoThread and more generally, all classes that have the same interface but a different implementation in the two execution modes are implemented using the Strategy pattern [GHJV95]. This makes adding other execution modes rather easy. As an example, Neko already supports two simulation engines, SimJava and a more lightweight simulation engine developed for Neko. One could imagine integrating further simulation packages.

[5] The reason is that most Java simulation packages are inspired by existing C or C++ packages and do not follow the Java philosophy of threading.

It must be noted that no restrictions apply if the application is only to be run in *one* of the execution modes. Thus distributed applications may use all features of Java and forget about NekoThreads, and simulations may exploit all features of the simulation package.

4.2 Networks

Neko networks constitute the lower layers of the architecture (see Fig. 4.1). The programmer specifies the network in the configuration file. No change is needed to the application code, not even if one changes from a simulated network to a real one or vice versa. In this section, we present real and simulated networks.

4.2.1 Real networks

Real networks are built on top of Java sockets (or other networking libraries). They use Java serialization for representing the content of NekoMessages on the wire.

TCPNetwork is built on top of TCP/IP. It guarantees reliable message delivery. A TCP connection is established upon startup between each pair of processes. UDPNetwork is built on top of UDP/IP and provides unreliable message delivery, which is sufficient for sending heartbeats in the example of Section 4.1.2. MulticastNetwork can send UDP/IP multicast messages. PMNetwork uses the PM portable communication library [THIS97] which bypasses the TCP/IP stack to take advantage of low-latency cluster interconnection networks, like Myrinet. Finally, EnsembleNetwork integrates the Ensemble group communication framework [VBH+98, Hay98] into Neko, to do reliable IP multicasts.

Neko is focused on constructing prototypes. Nevertheless, we performed some measurements to evaluate Neko's performance. We compared Neko's performance with the performance of Java sockets, using both TCP and UDP. According to [DAK00] the performance of Java and C sockets are rather close (within 5%) with the newest generation of Java Virtual Machines (JVMs), hence our comparison gives an indication of Neko's performance versus C sockets. We used the same benchmarks as [DAK00], from IBM's SockPerf socket micro-benchmark suite [IBM00], version 1.2.[6] The benchmarks are the following:

TCP_RR_1 A one-byte message (request) is sent using TCP to another machine, which echoes it back (response). The TCP connection is set up in advance. The result is reported as a throughput rate of transactions per second, which is the inverse of the round-trip time for request and response. With Neko, we use NekoMessages with null content; they still include 4 byte of useful data (type) and constitute the shortest messages we can send.

[6] These experiments do not benchmark all aspects of communication. Nevertheless, they should give an indication of the overhead imposed by Neko on the native sockets interface. The experiment CRR_64_8K was not performed, as it has no equivalent in Neko.

performance	Neko	Java	relative
TCP_RR_1 [1/s]	3836	8104	47%
UDP_RR_1 [1/s]	3756	8074	46%
TCP_STREAM_8K [kbyte/s]	7689	10861	71%

Table 4.1: Performance comparison of Neko and Java sockets.

UDP_RR_1 Same as TCP_RR_1, using UDP instead of TCP as a transport protocol.

TCP_STREAM_8K A continuous stream of 8 kbyte messages is sent to another machine, which continuously receives them. The reported result is bulk throughput in kilobytes per second.

We used two PCs running Red Hat Linux 7.2 (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 duplex Ethernet hub. The Java Virtual Machine was Sun's JDK 1.4.1_01. The absolute results, as well as the relative performance of Neko versus Java sockets, are summarized in Table 4.1. They show that Neko performance reaches at least 46% of Java performance in all tests, with TCP_STREAM_8K performance reaching 71%. Response times are more affected than throughput. The overhead is probably due to (1) serialization and deserialization of NekoMessages, (2) the fact that Java objects including NekoMessages are allocated on the heap and (3) the necessity of having a separate thread that maps from the send-receive communication mechanism of sockets to the send-deliver mechanism of Neko.

We will continue working on the performance optimization of Neko. The most promising directions are (1) to adapt standard Java serialization so that it performs well for short messages [PHN00] (it is optimized for dumping long streams of objects into a file) and (2) to improve its buffering strategy.

4.2.2 Simulated networks

Currently, Neko can simulate simplified versions of Ethernet, FDDI and CSMA-DCR [HLL98] networks. Complex phenomena, like collision in Ethernet networks, are not modeled, as they do not influence the network behavior significantly at low loads. The models are motivated and described in detail in [Ser98, TBW95]. Other models can be plugged into Neko easily, due to the simplicity of the network interface.

Simple complexity metrics can help predicting the performance of an algorithm. Several such metrics can be evaluated by simulation using Neko; the model behind each metric is implemented as a simulated network. The metrics include time complexity, which is roughly the number of communication steps taken by the algorithm, and message complexity, which is the number of messages gener-

ated by the algorithm [Lyn96, Sch97]. Neko was also helpful in evaluating the contention-aware metrics described in Chapter 5.

A different kind of simulated network proved to be useful in debugging distributed algorithms. The network delivers a message after a random amount of time, given by an exponential distribution. This network, although not particularly realistic in terms of modeling actual performance, usually "exercises" the algorithm more than an actual implementation. The reason is that this network tends to behave in a less deterministic way than a real network, as the message delays are independent and their variance is relatively high.

4.3 Algorithms

Neko was used for all the simulations and measurements in the remaining chapters of this thesis. The relevant parts of Neko are described in more detail there.

Group communication. Neko was used to develop various algorithms in the context of fault tolerant group communication (see Section 2.1.3.A and 2.2). They form the base of a future group communication toolkit. A list of algorithms implemented follows, grouped by the distributed problem they solve. A generic interface is defined for each problem.

- **Failure detectors** A heartbeat and a ping based failure detector are implemented, as well as a failure detector optimized for the Chandra-Toueg consensus algorithm that uses no failure detection messages. Two emulated failure detectors are included: the one detects an emulated crash after a fixed amount of time, and the other generates suspicions of correct processes at a certain rate, each suspicion lasting for a certain duration. They are mostly useful in simulation studies in which the failure detector is not modeled in detail. Finally, one of the failure detectors implements an alternative interface (Ω, see [CHT96]): instead of offering access to a list of crashed processes, the detector returns one process which is likely correct (the implementation returns the first process trusted by another underlying failure detector).

- **Consensus** Several published algorithms are implemented. Most are asynchronous and rely on failure detectors: [CT96] with some optimizations, [MR99], [Lam01] with some optimizations. The R-consensus algorithm relies on *weak ordering oracles* [PSUC02a] (see Section 2.1.3.B).

- **Reliable broadcast** Several algorithms from [FP01] are implemented.

- **Atomic broadcast** [DSU00] surveys and classifies most published atomic broadcast algorithms. Algorithms in the following classes are implemented: one uniform and one non-uniform fixed sequencer algorithm, one uniform

and one non-uniform privilege based algorithm, one communication history algorithm, and three destination agreement algorithms. Two of the destination agreement algorithms are fault tolerant: [CT96] (when used with a fault tolerant consensus algorithm) and [PSUC02a] (based on weak ordering oracles). The fixed sequencer algorithms [BSS91] have been integrated with a group membership service. The other algorithms do not tolerate crashes.

- **Group membership** One primary partition group membership algorithm is implemented [MS95]. It uses failure detectors to start view changes, and relies on consensus to agree on the next view.
- **Replication strategies** Active or state-machine replication [Sch90] of services is implemented. There is also an optimistic variant, in which the application is provided hints about the final order of requests, usually before the actual delivery of the request.

Benchmarks. The SockPerf benchmarks (see Section 4.2.1) are implemented and extended for multicast communication. Benchmarks for consensus and atomic broadcast algorithms (see Chapter 6) are implemented as well.

Other components. One component streams large arrays of values over the network as a sequence of relatively small messages. For instance, this component is useful when data recorded during a measurement session is collected: handling all the data as a single message is impractical because of the size of the data. Another component implements a "leaky bucket": it buffers outgoing messages to ensure that the rate of messages sent does not exceed a maximum rate.

Visualization tool. Neko contains a debugging tool that displays logs of messages graphically. The display options are configured using a GUI and/or an XML file.

4.4 Related work

Prototyping and simulation tools. The x-kernel and the corresponding simulation tool x-sim [HP91] constitute an object oriented C framework that has similarities with Neko. It is designed for building (lower level) protocol stacks. Efficient execution of the resulting protocols is a major goal. Instead, Neko's focus is easy prototyping of distributed protocols/applications. Neko is a much smaller framework that supports building the application as a hierarchy of layers, which resemble x-kernel protocol stacks, but in a simpler, more flexible way: (1) a layer is not restricted to adding/removing headers and fragmenting/reassembling messages, and (2) and a thread-per-layer (rather than thread-per-message) approach is possible. Moreover, Neko benefits from the advantages of using Java rather than C: serialization and deserialization of complex data and multithreading are supported.

The NEST simulation testbed [DSYB90] also supports prototyping to some extent. The code used for simulation is rather similar to UNIX networking code (in C): normally, only a few system calls have to be changed.

Microprotocol frameworks. Microprotocol frameworks aim at providing more flexible and ways for protocol composition than traditional ISO/OSI stacks. In these frameworks, protocol layers are only allowed to interact in well specified ways (e.g., a typical restriction is that layers cannot share state; they have to communicate using events that travel up or down the protocol stack). As a result, the code for protocols tends to be more maintainable than code built in a more ad hoc manner, and new protocols or protocols with a new set of properties can be composed easily. *x*-kernel, Ensemble [VBH+98], Coyote and Cactus [BHSC98, SH99] and Appia [MPR01] are examples of such systems, and these systems were used to construct group communication protocols, just like Neko. Neko also composes protocols out of layers, but as we have used it primarily for prototyping so far, we did not feel the need for the advanced architecture that microprotocol frameworks have. If the need arises, we plan to integrate some of the ideas behind these frameworks into Neko.

Simulators. There exist a variety of systems developed to simulate network protocols. Most of them concentrate on only one networking environment or protocol; a notable exception is NS-2 [FK00], where the goal is to integrate the efforts of the network simulation community. These tools usually focus on the network layer (and the layers below), with (often graphical) support for constructing topologies, detailed models of protocols and network components. Neko is focused on the application layer, rather than on support for constructing complex network models. We see the two directions as complementary: in order to obtain realistic simulations on detailed network models, Neko will have to be integrated with a realistic network simulator. The simplicity of Neko's network interface eases this task.

Message passing libraries. Neko (when used for prototyping) can be seen as a simplified socket library, with support for frequently occurring tasks like sending data structures, startup and configuration. A variety of simplified versions of the BSD C sockets interface are available (e.g., [ECM02]). However, they are at best as easy to use as the Java interface to sockets. Other message passing standards exist: MPI [SOHL+98, GHLL+98] and PVM [Sun90]. They focus on different aspects of programming than Neko: they are mostly used in high performance computing to implement parallel algorithms, and efficient implementation on massively parallel processors and clusters is crucial. The result is that their APIs are complex compared with Neko: they provide operations useful in parallel programming but hardly used in distributed systems: e.g., scatter, gather or reduce. The APIs tend to be complex also because they are C/Fortran style, even in Java implementations like mpiJava [BCFK99], jmpi [Din99], JPVM [Fer98] and jPVM [BS96].

Chapter 5

Contention-aware performance metrics

Resource contention is widely recognized as having a major impact on the performance of distributed algorithms. Nevertheless, the metrics that are commonly used to predict their performance take little or no account of contention. In this chapter, we define two performance metrics for distributed algorithms that account for network contention as well as CPU contention. We then illustrate the use of these metrics by comparing four atomic broadcast algorithms, and show that our metrics allow for a deeper understanding of performance issues than conventional metrics. We also validate the metrics using measurements on atomic broadcast algorithms running on a cluster.

The chapter is structured as follows. Section 5.2 presents related work. In Section 5.3, we present the system model on which our metrics are based. Section 5.4 presents a latency and a throughput metric. Section 5.5 presents a tool to help producing formulas for the metrics. Atomic broadcast algorithms used to illustrate and validate the metrics are presented in Section 5.6. We then illustrate the use of our metrics by comparing some algorithms in Section 5.7. Finally, the validation of the metrics follows in Section 9.

5.1 Introduction

State of the art: evaluating the performance of distributed algorithms. A variety of people are involved in the construction of distributed systems: algorithm designers invent and publish new algorithms, and engineers implement and deploy these algorithms. Both have a need for performance evaluation, though the techniques they use are different (performance evaluation techniques were introduced and contrasted in Section 3.3). Engineers need accurate results for a specific system, and have an implementation of the algorithm available, so they most often use simulations or measurements on a prototype. In contrast, algorithm designers invest considerable effort in proving the correctness of their algorithms (which is

essential) but often oversee the importance of predicting the performance of their algorithms. They most often use simple complexity metrics, which can be seen as very simple analytical models of performance, as these are easy to evaluate and give results meaningful for a wide range of systems. The price to pay is the low accuracy of the results. In fact, there is a wide gap between the results of the two classes of specialists, in terms of accuracy: the value of simple complexity metrics in predicting the performance of distributed algorithms is questionable. Let us next investigate some of the complexity metrics in detail.

Existing metrics for distributed algorithms. Performance prediction of distributed algorithms is usually based on two rather simplistic metrics: time and message complexity.

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

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

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

Contention-aware metrics. To solve the problem, we define two metrics (one latency metric and one throughput metric) which account for resource contention, both on the CPUs and the network. These metrics are still rather easy to evaluate: only one new parameter is introduced with respect to time and message complexity (see Section 5.3).

5.2 Related work

Resource contention in network models. Resource contention (also sometimes called congestion) has been extensively studied in the literature. The bulk of the publications about resource contention describe strategies to either avoid or reduce resource contention (e.g. [HP97, HYF99]). Some of this work analyze the performance of the proposed strategies. However, these analyses use models that are often specific to a particular network (e.g., [LYZ91]). Distributed algorithms are normally developed assuming the availability of some transport protocol. A metric that compares these algorithms must abstract out details that are only relevant to some implementations of a transport layer. In other words, it is necessary to relinquish precision for the sake of generality.

Resource contention in parallel systems. Dwork, Herlihy and Waarts [DHW97] propose a complexity model for shared-memory multiprocessors that takes contention into account. This model is very interesting in the context of shared memory systems but is not well suited to the message passing model that we consider here. The main problem is that the shared memory model is a high-level abstraction for communication between processes, and as such, it hides many aspects of communication that are important in distributed systems.

Computational models for parallel algorithms. Unlike for distributed algorithms, many efforts have been directed at developing performance prediction tools for parallel algorithms. However, the execution models are not adapted to distributed algorithms: for instance, the PRAM model (e.g., [Kro96]) requires that processors evolve in lock-steps and communicate using a global shared memory; the BSP model [Val90] requires that processors communicate using some global synchronization operation; the LogP model [CKP$^+$96] assumes that there is an absolute upper bound on the transmission delay of messages. These models are not adequate to predict the performance of distributed algorithms. The main reason is that they do not naturally suit distributed algorithms designed for the asynchronous system model, which do not assume any form of global synchronization nor any restriction on communication delays.

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

optimal reference is appealing, but this approach is orthogonal to the definition of a metric. The metric used is designed for the shared-memory model, and still ignores the problem of contention.

5.3 Distributed system model

The two metrics that we define in this chapter are based on an abstract system model which introduces two levels of resource contention: *CPU contention* and *network contention*. For easier understanding, we split the description of the model in two parts. In the first part, we give an overview of the model and sketch how distributed algorithms are executed in the model. In the second part, we give a complete detailed specification of the execution of algorithms. Finally, we illustrate the model with an example.

5.3.1 Overview of the model

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

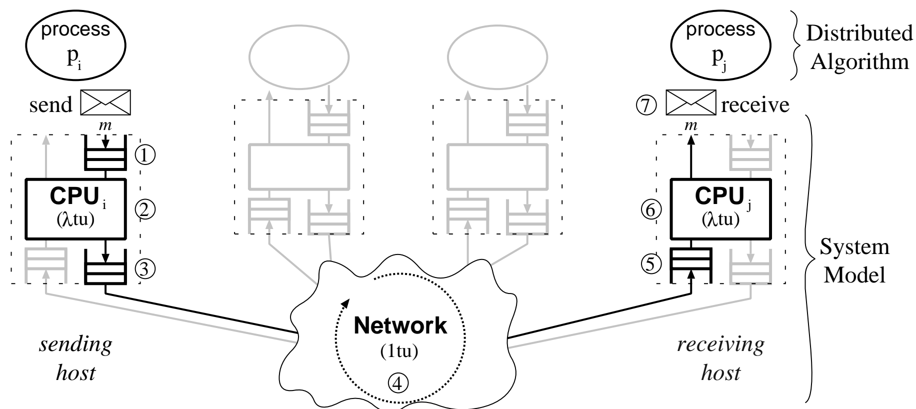


Figure 5.1: Decomposition of the end-to-end delay (tu=time unit).

The transmission of a message m from a sending process p_i to a destination process p_j occurs as follows (see Fig. 5.1):

1. m enters the *sending queue*[1] of the sending host, waiting for CPU_i to be available.

2. m takes the resource CPU_i for λ time units, where λ is a parameter of the system model ($\lambda \in \mathbb{R}^+$).

3. m enters the *network queue* of the sending host and waits until the network is available for transmission.

4. m takes the network resource for 1 time unit.

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

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

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

The parameter λ can be understood as the ratio of the amount of processing a message on a *CPU* resource to the amount of processing a message on the *network* resource. Being a ratio of positive quantities, it is a dimensionless quantity, ranging from 0 to infinity (bounds excluded).

5.3.2 Resource conflicts

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

In our description, each resource (CPU and network) is associated with one task, run concurrently with all other tasks. We now describe how these tasks resolve resource conflicts, and justify our decisions regarding the resolution of resource conflicts.

Network task. Concurrent requests to the network may arise when messages at different hosts are simultaneously ready for transmission. The access to the network is modeled by a round-robin policy,[2] illustrated by Algorithm 5.2.

We chose a round-robin policy as this is the simplest deterministic policy that is *fair*: if messages from several processes are waiting for transmission, this policy makes sure that each process gets an equal share of the network resource over a long time.

[1] All queues in the model (sending, receiving, and network queues) use a FIFO policy. For the sake of simplicity, queue sizes are *not* bounded: unlike real systems, the model has no built-in flow control.

[2] This was suggested by Jean-Yves Le Boudec.

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

Figure 5.2: Network access policy (executed by the network task).

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

We decided to give priority to outgoing messages because the equally simple alternative (priority to incoming messages) results in unacceptable behavior. Suppose that a process gathers messages from several sources but is only interested in the message arriving first: upon receiving this message, it sends a reaction message. If priority is given to incoming messages, the reaction is delayed until all incoming messages are processed. This behavior is not consistent with what a lot of distributed algorithms (e.g., replication algorithms) expect: outgoing messages are not delayed in such a way by incoming messages.

Interaction of tasks. At some values of λ (actually, rational values) the CPU and network tasks may want to take steps at exactly the same time: e.g., the network task may take a message from an outgoing queue when a CPU task puts a message into another outgoing queue. We want deterministic executions, therefore the order of taking these steps must be defined. We do so by assigning priorities to tasks: the network task has higher priority than process tasks. The choice is deliberate: its effect is as if messages spent $\lambda + \epsilon$ time units on a CPU resource, rather than λ, where ϵ is an infinitesimally small value. Consequently, algorithms do not exhibit a singular behavior in a system model with $\lambda = 0$; they behave at $\lambda = 0$ just like at small values of lambda (ϵ). A singular behavior at $\lambda = 0$ would not have any counterpart in reality: messages cannot be processed in a truly instantaneous way.

A similar problem arises when process tasks interact with the task(s) of the application. The solution is also similar: the process tasks have higher priority than the application tasks. Consequently, applications that react to incoming messages in time 0 and in time ϵ behave the same way; once again, there is no singular behavior corresponding to an application that reacts in a truly instantaneous way.

5.3.3 Messages sent to oneself

When describing an algorithm, it is often convenient to have processes send messages to themselves, or include themselves in the set of destinations of a message.

Such messages take neither CPU nor network resources in our model.

5.3.4 Multicast messages

Distributed algorithms often send a message to several processes. The way this is actually performed depends on the model (see below). Let process p send a message m to destinations $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$ where $i_1 < i_2 < \ldots < i_k$ and $k \geq 2$.

Definition 1 (point-to-point) *Model $\mathcal{M}_{\mathrm{pp}}(n, \lambda)$ is the model with parameters $n \in \mathbb{N}$ and $\lambda \in \mathbb{R}_0^+$, where $n > 1$ is the number of processes and λ is the relative cost between CPU and network. Multicasting is defined as follows: p sends the message m consecutively to all destination processes in lexicographical order: $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$.*

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

Definition 2 (broadcast) *Model $\mathcal{M}_{\mathrm{br}}(n, \lambda)$ is defined like $\mathcal{M}_{\mathrm{pp}}(n, \lambda)$, with the exception of multicast messages: p sends a single copy of m (rather than k copies), the network transmits a single copy of m (rather than k copies), and each destination process receives one copy of m (just like in $\mathcal{M}_{\mathrm{pp}}(n, \lambda)$).*

5.3.5 Illustration

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

Figure 5.3 shows the execution of this simple algorithm in model $\mathcal{M}_{\mathrm{br}}(3, 0.5)$. The upper part of the figure is a time-space diagram showing the exchange of messages between the three processes (message exchange, as seen by the distributed algorithm). The lower part is a more detailed diagram that shows the activity (send, receive, transmit) of each resource in the model. For instance, process p_1 sends a message m_1 to process p_2 and p_3 at time 0. The message takes the CPU resource of p_1 at time 0, takes the network resource at time 0.5, and takes the CPU resource of p_2 and p_3 at time 1.5. Finally, p_2 and p_3 simultaneously received m_1 at time 2.

Similarly, Fig. 5.4 shows the execution of the algorithm in model $\mathcal{M}_{\mathrm{pp}}(3, 0.5)$. The network is point-to-point, so whenever a message is sent to all, many copies of that messages are actually sent. For instance, process p_3 sends a copy of message m_3 to process p_1 (denoted $m_{3,1}$) at time 3. The message takes the CPU resource of p_3 at time 3, takes the network resource at time 4.5, and takes the CPU resource of p_1 at time 5.5. Finally, m_3 is received by p_1 at time 6.

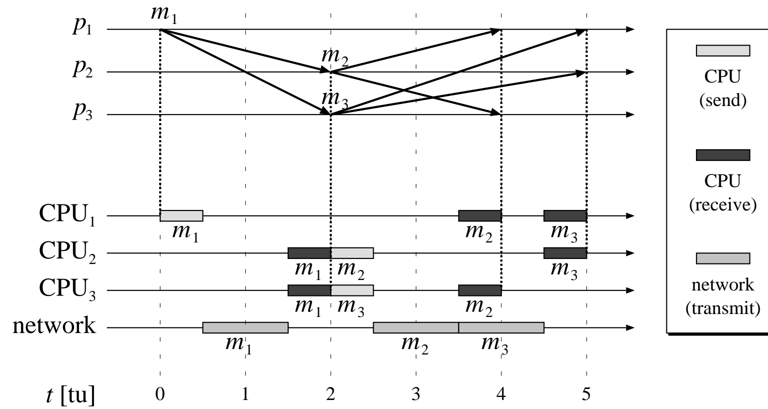


Figure 5.3: Simple algorithm in model $\mathcal{M}_{\mathrm{br}}(3, 0.5)$.

5.4 Definition of the contention-aware metrics

5.4.1 Latency metric

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

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

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

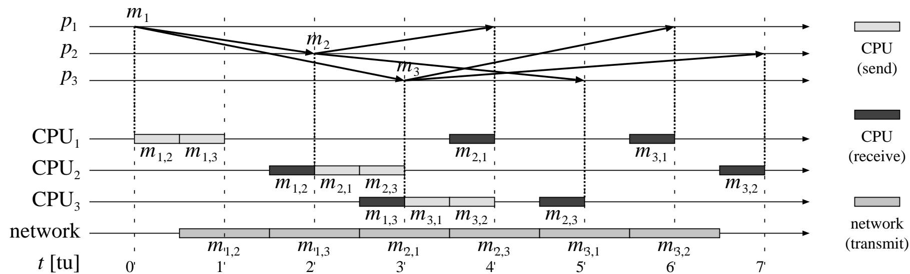


Figure 5.4: Simple algorithm in model $\mathcal{M}_{\mathrm{pp}}(3, 0.5)$ ($m_{i,j}$ denotes the copy of message m_i sent to process p_j).

Relation to complexity metrics. When $\lambda = 0$, that is, processing messages on the CPUs takes no time, $\text{Latency}(\mathcal{A})(n, \lambda)$ gives the number of messages exchanged by the algorithm $\mathcal{A}$ and is hence related to message complexity (it is not equal to message complexity, as message complexity is defined in a system model without contention).

When $\lambda \to \infty$, that is, processing messages on the network takes no time, $\lim_{\lambda\to\infty} \text{Latency}_{\text{br}}(\mathcal{A})(n, \lambda)/\lambda$ is related to time complexity. Consider an algorithm that executes in synchronous rounds: in each round, every process sends a message to every other process. Given such an algorithm, the above quantity is proportional to the number of rounds[3] , i.e., the latency degree of the algorithm $\mathcal{A}$ [Sch97].

5.4.2 Throughput metric

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

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

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

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

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

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

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

[3] n times the number of rounds in $\mathcal{M}_{\text{br}}(n, \lambda)$; every process performs one send and $n-1$ receive operations per round.

5.4.3 Mathematical properties of the metrics

The (latency or throughput) metric for a given (deterministic) algorithm running on a fixed number of processes is a function of λ; let us denote it $F(\lambda)$. It follows from the system model (see Section 5.3.2, interaction of tasks) that $F(\lambda)$ is right continuous: $\lim_{\lambda \to \lambda_0 + 0} F(\lambda) = F(\lambda_0)$. $F(\lambda)$ is not left continuous: jumps are still possible. Also, as all actions in the system model take either λ time units or 1 time unit (or another constant amount of time, if the algorithm has timeouts), each continuous piece of $F(\lambda)$ is a linear function of λ.

5.5 Tool to compute the metrics

Computing the contention-aware metrics by hand is often tedious, especially for the latency metric. Nevertheless, this method provides maximal insight: it produces parametric formulas with n and λ as parameters (and possibly other parameters associated with the algorithm). One can also compute the metrics by simulating the algorithm and the system model. However, this method only yields the metrics at fixed values of n and λ.

In this section, we describe a tool that provides more insight than plain simulation: it yields parametric formulas with λ. The formula is produced in a fully automated manner. The tool simulates the algorithm at certain fixed values of λ and records all events related to message passing. It then determines the interval $\lambda_1 \leq \lambda < \lambda_2$ in which the algorithm produces the same sequence of events. In this interval, the metric is a linear function of λ; the tool computes this function. These steps are repeated until formulas are found for the whole domain of λ ($\mathbb{R}_0^+$).

5.5.1 Building an activity network for an execution

Let us run a simulation of the algorithm in the system model $\mathcal{M}(n, \lambda)$ (that is, $\mathcal{M}_{\text{pp}}(n, \lambda)$ or $\mathcal{M}_{\text{br}}(n, \lambda)$) at some fixed value of $n = n_0$ and $\lambda = \lambda_0$. During the execution, we build the following graph:

- Nodes of the graph are *activities* occurring on any task (application process, CPU task or network task). There are input, output and local activities. The output activity send(m) and the input activity receive(m) correspond to passing a message with identifier m between tasks. Local activities of interest are those that take a certain time t to execute; other events take time 0. During the simulation, all these activities are recorded.

- Nodes are labeled by the amount of time that the corresponding activity takes. In our model, only the activity of processing messages takes time: 1 time unit on the network and λ_0 time units on the CPUs.

- Arcs of the graph correspond to *precedence relations* among activities. An activity can only start when all immediately preceding activities have finished. For any message m, send(m) precedes receive(m). Moreover, an

activity e precedes another activity e' if they are executed by the same task and e is executed before e'. Arcs are drawn when an activity is recorded: the new node has an arc coming from the node that corresponds to the previous activity on the task, and, if it is the activity receive(m), it also has an arc coming from the node corresponding to send(m).

Graphs of this type are known as *activity network* or PERT (Program Evaluation and Review Technique) chart. Using well-known techniques, it is possible to compute minimum completion times between any two activities, and many more things.

We use this activity network to compute the latency metric. Let us record the start and the end of the algorithm as activities. Then the latency metric is simply the minimum completion time between these two activities.

Of course, in this particular setting, one can compute the latency metric much easier, without constructing the activity network: during the simulation, one records the value of the simulation clock at the start and the end of the algorithm and then computes the difference. However, simulation only yields results for fixed values of λ, whereas the method based on the activity network is easily extended to yield parametric formulas. The extended method is described in the next section.

5.5.2 Parametric computations

Let us now label each node of the activity network with a parametric formula of the form $a + b \cdot \lambda$ rather than a scalar value. The latency metric $\mathrm{Latency}(\mathcal{A})(n_0, \lambda)$ is computed as before: it is the minimum completion time between the start and the end of the algorithm. The result is a linear function of lambda:

$$\mathrm{Latency}(\mathcal{A})(n_0, \lambda) = a_c + b_c \cdot \lambda \tag{5.1}$$

with some constants a_c and b_c. This formula is not only valid for $\lambda = \lambda_0$, the value used for the simulation that built the activity network. We now compute a value λ_1 and show that the formula is valid for any λ such that $\lambda_0 \leq \lambda < \lambda_1$, supposing that the algorithm $\mathcal{A}$ is deterministic.

On deterministic algorithms. We first precise what we mean by a deterministic algorithm. We modeled the system (the algorithm and the network; see Section 5.3) as a set of tasks interacting with instantaneous message passing; the interactions appear as activities in the activity network. A task is *deterministic* if it is fully described by a state machine, i.e., the state of the task only changes when it executes an activity, and the state of the task just before the activity and the activity itself determine the state just after the activity. The algorithm is deterministic if all of its tasks are deterministic.[4]

[4]This means that a deterministic algorithm cannot use random numbers or clocks. Neither can it have non-deterministic scheduling of its tasks, nor can it communicate with the outside world. If these assumptions are too restrictive, one can resort to probabilistic evaluation (for random numbers) or one can include a model of clocks, threading, and the outside world into the model.

If the algorithm is deterministic, the whole system is deterministic as the CPU and network tasks are deterministic (see Section 5.3.2). In a deterministic system, the relative order of activities on each task determines the whole execution. The activity network records the relative order of activities for $\lambda = \lambda_0$.

Computing λ_1. λ_1 is the value up to which one can increase λ without changing the relative order of activities on tasks. Formula 5.1 is thus valid in the range $\lambda_0 \leq \lambda < \lambda_1$. We compute λ_1 as follows. Let us compute the *slack* for each activity A_i ($i = 1, 2, \ldots, k$ where k is the number of activities), i.e., the amount of time for which A_i can be delayed without altering (increasing) the overall minimum completion time between the start and the end of the algorithm. All the slacks are parametric formulas; let us denote the slack for A_i by $a_s^i + b_s^i \cdot \lambda$. As we increase λ, the order of activities on each task remains the same as long as all slacks are non-negative:

$$\forall i = 1, \ldots, k \qquad a_s^i + b_s^i \cdot \lambda \geq 0$$

λ_1 is simply the highest value for which this holds. It is computed as follows:

$$\lambda_1 = \min_{i=1,\ldots,k} \left\{ \begin{array}{ll} -a_s^i / b_s^i & \text{if } b_s^i < 0 \\ +\infty & \text{if } b_s^i \geq 0 \end{array} \right\}$$

We still need to explain why λ_1 is strictly greater than λ_0. The reason is simply that we gave CPU tasks, i.e., the tasks that take λ time, the lowest priority of all (see Section 5.3.2): an activity of message processing on a CPU that finishes at some time T executes after all other activities that finish at time T, hence it is possible to increase λ by a small value without changing the order of activities.

5.5.3 Summary: the algorithm

The tool repeats the parametric computations described in the previous section until the latency metric is computed for all of the domain for λ. An overview of the algorithm is presented in Algorithm 5.5.

1: $\lambda_1 \leftarrow 0$
2: **repeat**
3: $\quad \lambda_0 \leftarrow \lambda_1$
4: $\quad$ simulate the algorithm A with $\lambda = \lambda_0$ and build the activity network
5: $\quad$ compute formulas for the overall completion time $a_c + b_c \cdot \lambda$ and the slacks
6: $\quad$ compute λ_1 using the slacks
7: $\quad$ output the result "latency is $a_c + b_c \cdot \lambda$ if $\lambda_0 \leq \lambda < \lambda_1$"
8: **until** $\lambda_1 = +\infty$

Figure 5.5: Overview of the algorithm that computes a parametric formula for the contention-aware metrics.

A few remarks regarding the algorithm are appropriate:

- In line 4, the algorithm only actually constructs the nodes for activities that take non-zero time.
- For a lot of problems, several different definitions are appropriate for the latency metric. E.g., for a broadcast algorithm, the start of the algorithm is the sending of a message, but the end of the algorithm can be the earliest delivery of the message, the latest delivery, etc. In line 4, the algorithm can compute several definitions of the latency metric in one go: as the completion time is computed for each activity, it is easy to compute each definition as the difference of the completion times of the start and end activities.
- The tool is not restricted to the latency metric. The throughput metric can be easily computed as well: in each simulation (line 3), one has to record the utilization of each resource (CPU and network resource), and use the utilizations to compute the throughput metric (in parametric form).

5.6 Atomic broadcast algorithms

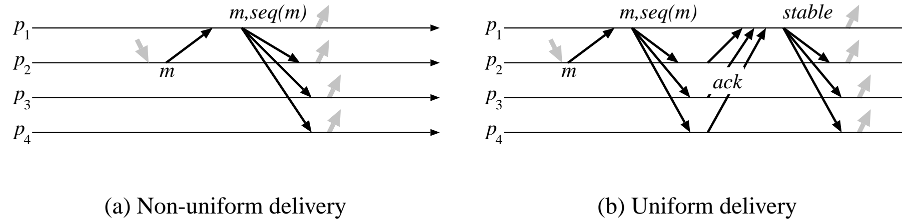
In this section, we describe the atomic broadcast algorithms used in the performance studies of Sections 5.7 and 9.

Atomic broadcast ensures that all destinations deliver the messages in the same order (see Section 2.2.3). Our previous work [DSU00] classifies more than 60 different algorithms into four classes, based on differences in the ordering mechanism: *sequencer*,[5] *privilege-based*, *communication history*, and *destinations agreement*. For the performance studies, we chose representative algorithms of each class. The representative algorithms are simplified variants of existing algorithms. Whenever possible, we consider both a uniform and a non-uniform variant. For readability and conciseness, we give only an informal description of each algorithm, illustrating their execution on a time-space diagram. Each scenario illustrates an execution of the algorithm wherein a single process broadcasts one message m only. We also assume that no failure occurs during the execution. The communication pattern thus described is sufficient for computing the contention-aware metrics. The pseudo-code of each algorithm is given in the Appendix, in Section A.2.

5.6.1 Sequencer

In sequencer algorithms, one process or several processes are elected as sequencers that are responsible for ordering all messages. [DSU00] distinguishes *fixed sequencer* algorithms with one sequencer process from *moving sequencer* algorithms in which the role of sequencer passes from one process to another. We chose two simple fixed sequencer algorithms to represent this class (moving sequencer algorithms are discussed later on).

[5] Actually, [DSU00] distinguishes *fixed sequencer* and *moving sequencer* algorithms.



(a) Non-uniform delivery

(b) Uniform delivery

Figure 5.6: Representative for sequencer algorithms.

The representative algorithms run as follows (see Fig. 5.6): when a process p wants to broadcast a message m, it sends m to the sequencer. The sequencer assigns a sequence number to m, and sends both m and the sequence number to the other processes. In the non-uniform algorithm, processes deliver m as soon as they receive it with its sequence number. In the uniform algorithm, the processes can deliver m only after it has been acknowledged by all processes, as shown on Fig. 5.6(b).

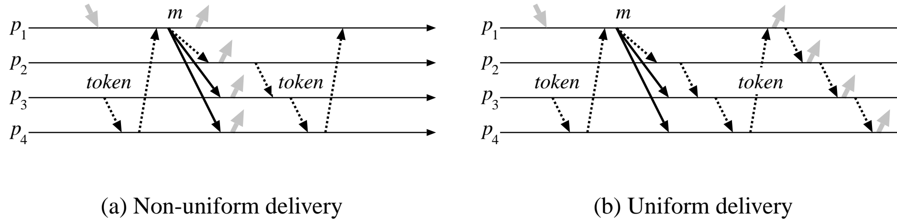
Fixed sequencer algorithms described in the literature are rarely uniform. The main reason is probably because the cost of uniformity is comparatively higher for fixed sequencer algorithms than for those of other classes. Isis [BSS91] and Amoeba [KT91b] are two well-known examples of fixed sequencer algorithms.

Moving sequencer algorithms are not represented in our studies. As the role of sequencer is passed from one process to another by means of a token, the communication pattern of moving sequencer algorithms is very close to that of privilege-based algorithms, described in the next section. For this reason, analyzing such algorithms would not yield a lot of additional information. The Atomic broadcast algorithm proposed by Chang and Maxemchuck [CM84] is a well-known example of a moving sequencer algorithm.

5.6.2 Privilege-based

With privilege-based algorithms, the delivery order is determined by the senders. Whenever a process has a message to broadcast, it must first obtain the privilege to do so.

The non-uniform algorithm works as follows (see Fig. 5.7). A token carrying a sequence number constantly circulates among the processes. When a process p wants to broadcast a message m, it simply stores m in a send queue until it receives the token. When p receives the token, it extracts m from its send queue, uses the sequence number carried by the token, and broadcasts m with the sequence number. Then, p increments the sequence number and transmits the token to the next process. To reduce the number of messages, p can broadcast the token along with m in a single message. When a process receives m, it delivers m according to its sequence number.



(a) Non-uniform delivery (b) Uniform delivery

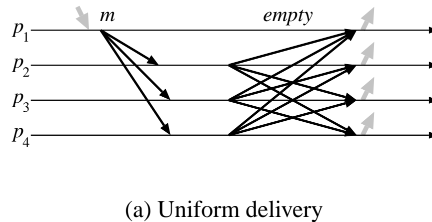
Figure 5.7: Representative for privilege-based algorithms.

In the uniform algorithm (Fig. 5.7(b)), the token also carries the acknowledgments. Before delivering a message m, processes must wait until they know that m is received by all, which requires a full round-trip of the token.

Totem [AMMS$^+$95] is a typical illustration of a privilege-based Atomic broadcast algorithm.

5.6.3 Communication history

With communication history algorithms, the delivery order is determined by the senders, just like with privilege-based algorithms. The difference is that processes can send messages at any time. The destinations observe the messages generated by the other processes to learn when delivering a message will no longer violate the total order.



(a) Uniform delivery

Figure 5.8: Representative for communication history algorithms.

The algorithm that we consider in this paper (see Fig. 5.8) works as follows. A partial order is generated by using logical clocks [Lam78] to "timestamp" each message m with the logical time of the *A-broadcast*(m) events. This partial order is then transformed into a total order by using the identifier of sending processes to arbitrate concurrent messages as follows: if two messages m and m' have the same logical timestamp, then m is before m' if the identifier of $sender(m)$ is smaller than the identifier of $sender(m')$. It follows that a process p can deliver some message m only once it knows that no message m' received in the future will carry a lower timestamp (or an equal timestamp and a smaller identifier for $sender(m')$

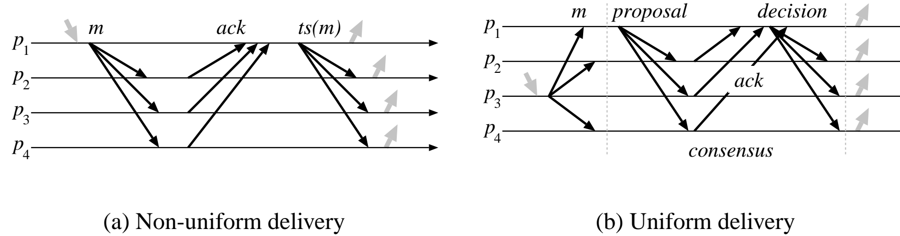
than the identifier of p). The algorithm requires FIFO channels (which are easily implemented by attaching sequence numbers to messages).

The algorithm as described so far would not be live because a silent process would prevent other processes from delivering. To avoid this, a process p is required to broadcast an *empty* message after a delay Δ_{live}, if it has nothing else to broadcast. When computing the latency metrics, we make the simplifying assumption that $\Delta_{\text{live}} = 0$. In the case of computing the throughput, the scenario does not generate any empty message anyway.

Communication history algorithms are basically an application of Lamport's mutual exclusion algorithm based on logical clocks [Lam78]. Some examples are given by Psync [PBS89], and Newtop [EMS95].

5.6.4 Destinations agreement

With destinations agreement algorithms, the delivery order is determined by the destination processes. In short, this is done in one of two ways; (1) ordering information generated by every process is combined deterministically, or (2) the order is obtained by an agreement between the destinations (e.g., consensus).



(a) Non-uniform delivery

(b) Uniform delivery

Figure 5.9: Representative for destinations agreement algorithms.

The non-uniform representative algorithm uses the first approach and is adapted from Skeen's algorithm, as described by Birman and Joseph [BJ87]. The algorithm works as follows (see Fig. 5.9(a)). To broadcast a message m, process p_1 sends m to all processes and acts as a coordinator for the delivery of m. Upon receiving m, a process q sends an acknowledgment and a logical timestamp $ts_q(m)$ back to p. Process p_1 gathers all timestamps and computes the final timestamp $TS(m)$ as the maximum of all received timestamps. Finally, p_1 sends $TS(m)$ to all processes which deliver m according to $TS(m)$.

The uniform representative algorithm uses the second approach and is due to Chandra and Toueg [CT96]. The algorithm works as follows (see Fig. 5.9(b)). Messages are sent to all processes without any additional information. Whenever the algorithm has messages to order, it runs a consensus algorithm to decide on the order (see Section 2.2.1). We use the Chandra-Toueg $\Diamond S$ consensus algorithm [CT96] for this purpose: one process (called *coordinator*) proposes an order, waits for acknowledgments from a majority of all processes (these are all positive if no

process is suspected to have crashed), and then broadcasts its decision to all. Upon receiving the decision, the processes deliver the messages concerned. A detailed pseudocode description of the algorithm is given in Section B.2, in the Appendix.

5.7 Comparison of atomic broadcast algorithms

We now illustrate the use of our two metrics by comparing four different atomic broadcast algorithms described in Section 5.6, using the contention-aware metrics, as well as time and message complexity. The four algorithms are the non-uniform sequencer algorithm (called Seq in the sequel; see Fig. 5.6(a)), the uniform privilege-based algorithm (PB; see Fig. 5.7(b)), the uniform communication history algorithm (CH; see Fig. 5.8(a)), and the non-uniform destinations agreement algorithm (DA; see Fig. 5.9(a)). We show that our metrics yield results that are more precise than what can be obtained by relying solely on time and message complexity. This confirms the observation that contention is a factor that cannot be overlooked.

The purpose of this section is to illustrate the use of the contention-aware metrics; the actual results of the comparison are of less interest. The reason is that in order to obtain truly useful results, one would have to carefully choose algorithms with more similar properties: the four algorithms differ not only with respect to uniformity, but also regarding that some of them are easily adapted to open groups, where senders can be outside of the destination group, while others only work in closed groups (just to name another difference). Also, one would need to examine them under a variety of workloads and faultloads. Hence a truly useful comparison would require a much more extensive study. In fact, such extensive studies with comparable algorithms are the subject of Chapters 7 and 8.

An extension of the study described in this section, comprising a uniform and a non-uniform variant for each class of atomic broadcast algorithms [DSU00] and using the same metrics, appears in [Déf00].

5.7.1 Latency metric

We now analyze the latency of the four atomic broadcast algorithms. The workload is such that a single process broadcasts one message m only. We also assume that no failures occur. The latency of the algorithm with respect to a message m is defined as follows: the algorithm starts when a process executes A-Broadcast(m) and ends when the last process executes A-Deliver(m).[6] Only the PB algorithm (Fig. 5.7(b)) constituted a special case: the latency of this algorithm depends on where the token is at the moment of executing A-broadcast(m). We computed the latency supposing that the token arrives after its expected arrival time, that is, one half of the time needed for a full round-trip.

[6]This definition corresponds to *late latency*, introduced in Section 6.1.

The results for the model $\mathcal{M}_{\text{pp}}(n, \lambda)$ are summarized in Table 5.1 and compared in Fig. 5.10(a).[7] Table 5.1 also shows the time complexity of the algorithms. For time complexity, we use the *latency degree* [Sch97]: roughly speaking, an algorithm with latency degree l requires l communication steps.

Alg. $\mathcal{A}$	$\text{Latency}_{\text{pp}}(\mathcal{A})(n, \lambda)$		**Time complexity**
Seq	$2(2\lambda + 1) + (n - 2)\max(1, \lambda)$		2
PB	$(2.5n - 1)(2\lambda + 1) + \max(1, \lambda)(n - 1)$		$2.5n - 1$
CH	$\approx 3(n-1)\lambda + 1$	if $n \leq \lambda + 2$	2
	$\approx \frac{1}{2}n(n-3) + 2\lambda n + \frac{1}{2}\lambda^2 - \frac{3}{2}\lambda$	if $n \leq 2\lambda + 3$	
	$\approx \frac{1}{2}n(n-1) + 2\lambda n + \lambda^2 - \frac{7}{2}\lambda - 3$	if $n \leq 4\lambda - 4$	
	$\approx n(n-1) + \lambda^2 + \lambda + 5$	otherwise	
DA	$\approx 3(n-1) + 4\lambda$	if $\lambda < 1$	3
	$\approx (3n-2)\lambda + 1$	if $\lambda \geq 1$	

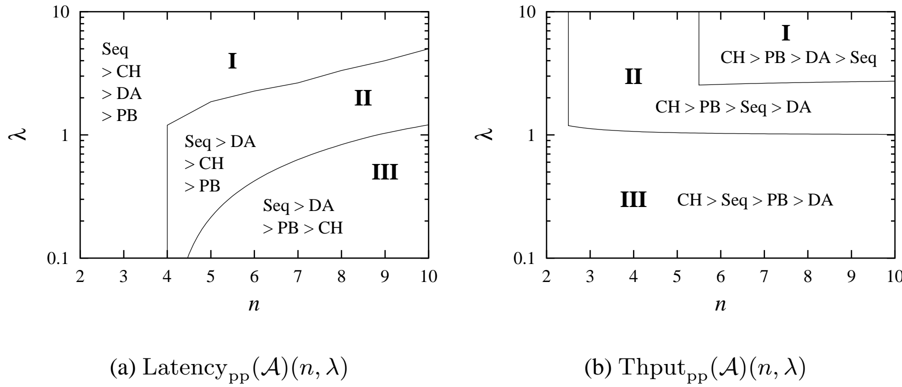
Table 5.1: Latency metric: evaluation of atomic broadcast algorithms (in model $\mathcal{M}_{\text{pp}}(n, \lambda)$).

Figure 5.10(a) represents the results of the comparison between the four algorithms with respect to the latency metric. The area is split into three zones (I, II and III) in which algorithms perform differently with respect to each other (e.g., in Zone I, we have Seq $>$ CH $>$ DA $>$ PB, where $>$ means "better than"). The latency metric and time complexity yield the same results for three of the four algorithms: Seq, PB and DA. Both metrics yield that Seq performs better than DA, which in turn performs better than PB. For CH, time complexity (Table 5.1) suggests that it always performs better than the other algorithms. This comes in contrast with our latency metric which shows that the relative performance of CH are dependent on the system parameters n and λ. The reason is that CH generates a quadratic number of messages and is hence subject to network contention to a greater extent. Time complexity is unable to predict this as it fails to account for contention.

5.7.2 Throughput metric

We now analyze the throughput of the four algorithms. In a throughput analysis, one run of the algorithm should not be considered in isolation. Indeed, many algorithms behave differently whether they are under high load or not (e.g., CH does not need to generate null messages under high load). For this reason, the throughput metric is computed by considering a run of the algorithm *under high load*. We also assume that every process atomically broadcasts messages, and that the emission

[7] For reasons of clarity, we choose to give approximate formulas for $\text{Latency}_{\text{pp}}(\text{CH})(n, \lambda)$ and $\text{Latency}_{\text{pp}}(\text{DA})(n, \lambda)$. The expressions given for these two algorithms ignore a factor that is negligible compared to the rest of the expression. The exact expressions are given in the appendix, Section A.3. A description of the analysis is given in [UDS00a].

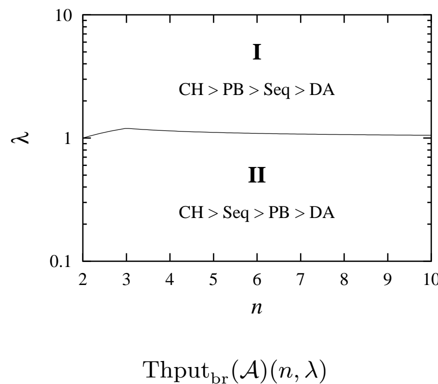


(a) $\mathrm{Latency}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$ (b) $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$

Figure 5.10: Comparison of atomic broadcast algorithms in the point-to-point model ($\mathcal{A} > \mathcal{A}'$ means $\mathcal{A}$ "better than" $\mathcal{A}'$).

is distributed fairly among them. For each algorithm, we compute the value of the throughput metric in model $\mathcal{M}_{\mathrm{pp}}(n, \lambda)$. The results are summarized in Table 5.2. The full description of the analysis is given in [UDS00a].

Figure 5.10(b) illustrates the relative throughput of the four algorithms. The graph is split into three zones (I, II and III) in which algorithms perform differently with respect to each other. The throughput metric and message complexity both yield that CH performs better than PB which in turn performs better than DA. However, the two metrics diverge when considering Seq. Indeed, while message complexity (Table 5.2) suggests that Seq always performs better than PB and DA, our throughput metric shows that it is not always the case. In fact, Seq is more subject to CPU contention than the other three algorithms. This type of contention



$\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$

Figure 5.11: Comparison of atomic broadcast algorithms in the broadcast model ($\mathcal{A} > \mathcal{A}'$ means $\mathcal{A}$ "better than" $\mathcal{A}'$).

Algorithm $\mathcal{A}$	$(\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n,\lambda))^{-1}$	**Message complexity**
Seq	$(n-\frac{1}{n})\cdot\max(1,\lambda)$	$n-\frac{1}{n}$
PB	$n\cdot\max(1,\frac{2\lambda}{n})$	n
CH	$(n-1)\cdot\max(1,\frac{2\lambda}{n})$	$n-1$
DA	$3(n-1)\cdot\max(1,\frac{2\lambda}{n})$	$3(n-1)$

Table 5.2: Throughput metric: evaluation of atomic broadcast algorithms (in model $\mathcal{M}_{\mathrm{pp}}(n,\lambda)$).

is especially noticeable in systems with large values of λ. Message complexity fails to pinpoint this, as it does not take CPU contention into account.

5.7.3 Latency and throughput in broadcast networks

The analyses in model $\mathcal{M}_{\mathrm{br}}(n,\lambda)$ are not much different. The full description of the analysis is given in [UDS00a]. In fact, there are fewer messages and less con-

Algorithm $\mathcal{A}$	$\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$
Seq	$2(2\lambda+1)$
PB	$(\frac{5n}{2}-1)(2\lambda+1)+\max(1,\lambda)$
CH	$4\lambda+n$
DA	$6\lambda+3+(n-2)\cdot\max(1,\lambda)$

Table 5.3: $\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$: evaluation of atomic broadcast algorithms.

Algorithm $\mathcal{A}$	$(\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda))^{-1}$	**Msg complexity**
Seq	$\frac{2n-1}{n}\max(1,\lambda)$	$2-\frac{1}{n}$
PB	$\max(2,\frac{n+2}{n}\lambda)$	2
CH	$\max(1,\lambda)$	1
DA	$\max(n+1,\frac{4n+1}{n}\lambda)$	$n+1$

Table 5.4: $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$: evaluation of atomic broadcast algorithms.

tention. Table 5.3 and Table 5.4 show the results of the two metrics in a broadcast network ($\mathrm{Latency}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$ and $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n,\lambda)$). Apart from the fact that these results are simpler than in a model with point-to-point communication, there are interesting differences.

According to the latency metric, for any value of λ and n, the algorithms are always ordered as follows:

$$\mathrm{Seq} > \mathrm{CH} > \mathrm{DA} > \mathrm{PB}$$

Unlike the results obtained with $\mathrm{Latency}_{\mathrm{pp}}(\mathcal{A})(n,\lambda)$, there is only one single zone with a broadcast network (hence not shown in Fig. 5.11). This zone corresponds

to zone I depicted on Figure 5.10(a) but, in model $\mathcal{M}_{\mathrm{br}}(n, \lambda)$, the algorithms are not ordered differently as n increases. This is easily explained by the fact that CH generates a quadratic number of messages in model $\mathcal{M}_{\mathrm{pp}}(n, \lambda)$ while it generates a linear number of messages in model $\mathcal{M}_{\mathrm{br}}(n, \lambda)$. The latency of the three other algorithms is not so much different in the two models because they generate a linear number of messages in both models.

Similarly, $\mathrm{Thput}_{\mathrm{br}}(\mathcal{A})(n, \lambda)$ yields simpler results than $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$. As shown in Figure 5.11, the parameter space is cut into two zones I and II (instead of three for $\mathrm{Thput}_{\mathrm{pp}}(\mathcal{A})(n, \lambda)$, as shown on Fig. 5.10(b)). The difference between the two zones is the relative performance (throughput) of Seq and PB. This yields that PB is better than Seq when the CPU is a limiting factor. In fact, Seq is limited by the sequencer process which becomes a bottleneck. Conversely, PB spreads the load evenly among all processes, and thus no process becomes a bottleneck. Once again, both classical metrics (time and message complexity) fail to capture this aspect of the algorithms' execution.

5.8 Experimental validation

In this section, we present the results of the validation of the contention-aware latency metric by measurements performed on a local area network. For each of the seven atomic broadcast algorithms described in Section 5.6 (four of which were also used in Section 5.7) we (1) measured the latency, (2) computed the contention-aware latency and (3) computed some traditional complexity metrics. We then used regression models to estimate the values of the parameters of the metrics. Our results show that the contention-aware metric is a better estimate of measured values than traditional complexity metrics, according to the sums of squared errors given by the regression models.

5.8.1 Workloads and metrics

In our experiments, we vary the number of participating processes n from 2 to 11. We consider isolated broadcasts: all messages belonging to a given broadcast are transmitted before a new broadcast starts (with the measurements, sending broadcasts every 10 ms was enough to guarantee this). Each broadcast is sent to all processes. Broadcasts are sent by each process using a round-robin scheme. No failures and no failure suspicions occur during the experiments.

For each broadcast, we measure the time that elapses from the time of sending to the time of the i-th delivery, for all $i = 1, \ldots, n$ (that is, n latency values are recorded for each broadcast). The time until the first delivery is called *early latency*, and the time until the last delivery is called *late latency*.[8] All latency values are averaged over all executions and the set of senders.

[8] See Section 6.1 for more verbose definitions of latency metrics.

Altogether, we evaluate 7 algorithms, with $n = 2$ to 11 processes and n latency metrics for each value of n. Our data set comprises thus $7 \cdot (2 + \ldots + 11) = 455$ data points.

5.8.2 Computing the metrics

We computed the contention-aware latency metric using the tool described in Section 5.5. We ran the tool once for each combination of algorithm, number of processes and possible sender. We then averaged the latency formulas over the set of senders. Both the point-to-point model ($\mathcal{M}_{\mathrm{pp}}(n, \lambda)$) and the broadcast model ($\mathcal{M}_{\mathrm{br}}(n, \lambda)$) were used.

Simpler complexity metrics (time and message complexity) have been computed manually.

Finally, we would like to point out that the privilege-based algorithms needed special treatment. Contrary to all other algorithms, the privilege-based algorithms generate messages even when no atomic broadcast is taking place (the token is passed around; see Fig. 5.7). For this reason, the tool that computes the latency metric cannot detect the end of an atomic broadcast; it usually detects the end of a broadcast by noticing that there are no messages are in transit on the simulated network. To cope with this problem, we had to (1) generate the token at the sending process when a broadcast begins, (2) remove the token after three round-trips, and (3) add half of the token round-trip time, i.e., the mean time until the token arrives at a given process, to the computed latency formulas.

5.8.3 Measurements

We used the TCP protocol to implement the algorithms. We chose TCP as all algorithms require reliable channels. As future work, we would like to repeat the measurements using a lightweight reliable multicast layer on top of IP multicast.

The duration of each experiment was 2000 s; as broadcasts are sent every 10 ms, this corresponds to 200'000 broadcasts. We need such long experiments because the Java just-in-time (JIT) compiler only acts after a large number of broadcasts: the latency only stabilizes after about 20'000-50'000 broadcasts. For this reason, we discard the first one third of the measured values. We also drop outliers, defined as being at least 5 times as much as the mean latency. Most outliers seem to be due to Java garbage collection; a garbage collection run shows up as an increasing sawtooth in the scatter plot (we ignore garbage collection as (1) it is unpredictable and (2) its overhead could easily be decreased by careful memory allocation policies). In each run, less than 2% of all data are dropped. As the remaining data still shows long term correlations (over minutes), we computed 95% confidence intervals from 3 independent experiments for each possible setting (algorithm and number of processes).

5.8.4 Analyzing the data

Predicting latency with the contention-aware metric. In order to relate the predictions of the contention-aware metric to the measured values, we need to determine the parameter λ, as well as the time unit of the metric, denoted by t_n: the time that a message spends on the network resource. We could determine these parameters by (1) measuring the latency of two simple algorithms, (2) setting the measured values equal to the corresponding formulas given by the contention-aware metric, and (3) solving the resulting set of equations for λ and t_n. Unfortunately, the contention-aware metric is not a sufficiently accurate description of reality to make this simple approach work: it abstracts away a lot of differences in the execution of algorithms, e.g., the length and the structure of messages (the more complex the structure of a message, the more time it takes to serialize and deserialize the message).[9] Therefore we need to use a more indirect way of determining the parameters λ and t_n: we build a regression model to fit the predictions to all measured data points, in which λ and t_n appear as predictor variables.

The regression model for the contention-aware metric is as follows. Suppose for a moment that the latency formula is given by $a \cdot \lambda + b$ over the whole range of λ, with some constants a and b. Then we can use the following simple regression model to get λ and t_n:

$$L = (a\lambda + b) \cdot t_n + L \cdot e$$

where L is the measured value, and e is the relative error of estimating the latency.

The model can be rewritten as follows:

$$1 = \frac{a}{L} \cdot (\lambda \cdot t_n) + \frac{b}{L} \cdot t_n + e$$

In this form, it is more visible that the model is a regression model with two predictor variables $\lambda \cdot t_n$ and t_n. We used well-known techniques to find values for the predictor variables that minimize the sum of squared errors (SSE) for e over all data points. The basic assumptions for using such a regression technique hold: in particular, it is safe to assume that the magnitude of e values is constant, or, in other words, that the magnitude of the absolute error of estimating the latency ($L \cdot e$) is proportional to the measured latency L.

In the general case, the latency formula is not linear over the whole range of λ, as we supposed so far. It is a piecewise linear function of λ, as illustrated in Fig. 5.12 (see also Section 5.4.3). Just as before, we are looking for the λ value that gives the least sum of squared errors (SSE) globally. We do so by performing a regression for each of the pieces, with different values for a and b in the regression model. Then we check if the λ computed falls within the right interval. If it does (lines A and C in Fig. 5.12) then it is a candidate that might yield the global minimum for the SSE. If it does not (lines B and D in Fig. 5.12) then the bounds of

[9]The results in Section 5.8.5 show what accuracy can be achieved with the contention-aware metric.

the interval for λ are candidates.[10] The SSE values that each candidate yields are compared to find the global minimum for the SSE.

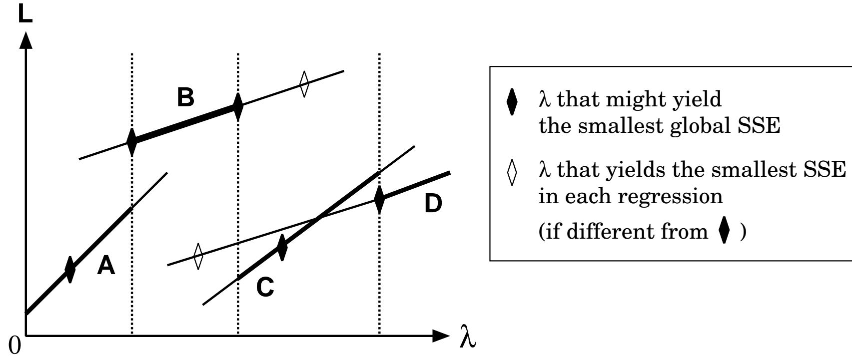


Figure 5.12: Latency as a function of λ in the general case.

We performed the above regression study for both network models: $\mathcal{M}_{\mathrm{pp}}(n, \lambda)$ and $\mathcal{M}_{\mathrm{br}}(n, \lambda)$.

Predicting latency with traditional complexity metrics. We consider three metrics: time complexity, and two kinds of message complexity: given a multicast message to k destinations, we consider this message as k messages in the first kind, and as one message in the other kind.

Fitting the predictions of any of these complexity metrics to the measured values only requires one time parameter, denoted by t. Just as before, we determine this parameter using regression. The regression model is the following:

$$L = c \cdot t + L \cdot e \tag{5.2}$$

where c is the value of the complexity metric, and t is the only predictor variable. Only one regression model is necessary.

Comparing metrics. We compare the metrics (both contention-aware and traditional) using the SSE that the regression yields for each. Lower values indicate that the metric in question is better at predicting the latency.

Note that the contention aware-metrics have two parameters, whereas traditional complexity metrics have only one parameter. Hence a direct comparison of the contention-aware metrics and the traditional complexity metrics might seem unfair: the flexibility provided by the extra parameter might partially explain why the contention-aware metrics yield better predictions. In order to have a fair comparison, we also consider the linear combination of time and message complexity.

[10] Actually, the explanation and Fig. 5.12 is simplified for better readability. One might also need to consider other values of λ. The reason is that the SSE as a function of λ might have several local minima, as λ is computed as a ratio of two regression variables.

Metric	s_e	Parameters of metric
contention-aware, $\mathcal{M}_{\mathrm{pp}}(n,\lambda)$	**0.131**	$\lambda = 4.01$ $t_n = 35.5\,\mu$s
contention-aware, $\mathcal{M}_{\mathrm{br}}(n,\lambda)$	**0.201**	$\lambda = 0.187$ $t_n = 267\,\mu$s
time complexity	**0.582**	$t = 590\,\mu$s
message complexity (pp)	**0.595**	$t = 72.1\,\mu$s
message complexity (br)	**0.403**	$t = 293\,\mu$s
time and message complexity (pp) combined	**0.449**	$t_{tc} = 368\,\mu$s $t_{mc} = 43.1\,\mu$s
time and message complexity (br) combined	**0.403**	$t_{tc} = -41.5\,\mu$s $t_{mc} = 309\,\mu$s

Table 5.5: Estimating the latency of algorithms using a variety of metrics using regression. s_e denotes the standard deviation of errors; the lower, the better.

The resulting metrics have two parameters, just like the contention-aware metric. The regression model is then the following (compare with Formula 5.2):

$$L = c_{tc} \cdot t_{tc} + c_{mc} \cdot t_{mc} + L \cdot e$$

where c_{tc} and c_{mc} are the values of the time and message complexity metric, respectively, and t_{tc} and t_{mc} are the corresponding predictor variables. The SSE obtained from this model is directly comparable with the SSE obtained from the contention-aware metric. Note that this model is by no means inspired from reality, unlike the complexity metrics and the contention-aware metric; it is just a mathematical construction.

5.8.5 Results

Our main results are summarized in Table 5.5. Similarly to the contention-aware metrics, the table shows two variants of message complexity: the point-to-point (pp) variant considers a multicast message as multiple point-to-point messages, whereas the broadcast (br) variant considers it as one message. s_e, the standard deviation of relative errors, is proportional to the SSE and is thus a lower-is-better measure of the goodness of the regression. Comparing the s_e values, we can make a number of observations:

- All of the traditional metrics do much worse at predicting latency than the contention-aware metric in either $\mathcal{M}_{\mathrm{pp}}(n,\lambda)$ or $\mathcal{M}_{\mathrm{br}}(n,\lambda)$, including the models with the same number of parameters, namely the combinations of time and message complexity. This is the main result of this study. It shows that resource contention is a phenomenon worth modeling.

- $\mathcal{M}_{\mathrm{pp}}(n,\lambda)$ yields more accurate predictions than $\mathcal{M}_{\mathrm{br}}(n,\lambda)$. This is hardly surprising, as we used the point-to-point TCP protocol in the measurements, and thus we could not exploit the multicast support at the network level.

Hence we expected that the model with only point-to-point messages is a better fit.

- Of all traditional metrics, time complexity gives the second worst results. It is significantly surpassed by the broadcast variant of message complexity, and is not much better than the point-to-point variant. This is surprising because, unlike time complexity, message complexity is usually not thought of as a latency metric.

The regression techniques used depend on the assumption that the measured values have been determined with no error. In practice, this means that the errors of measurement should be much smaller than the error of the regression. In our study, this assumption holds: the average relative error, computed from the 95% confidence intervals for the measured values, is 0.0405, much smaller than any of the s_e values in Table 5.5.

Graphs. In the rest of the section, we present some graphs to illustrate our results (Figures 5.13 to 5.16). Each graph shows the relative error of the latency on the vertical axis, i.e., the difference of the values estimated with a metric and the measured value, divided by the measured value. The zero horizontal axis is given as a reference; the closer a curve is to the reference axis, the better. The horizontal axis shows the number of processes. Each row shows results for one algorithm only. The left and right columns show the relative errors of the early and the late latency, respectively.

Figures 5.13 and 5.14 show the relative error of the latency estimates obtained with five different metrics: all metrics from Table 5.5, except for standalone message complexity. The purpose of the graphs is to show visually that the contention-aware metrics yield better results than conventional complexity metrics, for nearly all of the atomic broadcast algorithms. Comparing graphs with different algorithms is less interesting, hence the vertical scales are different in each graph in order to make the curves fill up all available space.

The second set of graphs (see Figures 5.15 and 5.16) show only the relative error of the latency estimated by the contention-aware point-to-point metric, the best match in Table 5.5. The vertical scales are the same in all graphs. One can make a number of observations:

- One can see that the errors are systematic rather than random. This is not surprising, as the contention-aware metric relies on a simple model.

- The metric systematically overestimates the latency of some algorithms (see e.g., the uniform destinations agreement algorithm in Fig. 5.16) and underestimates the latency of some other algorithms (see e.g., the uniform sequencer algorithm in Fig. 5.15). This seems to be correlated to how simple the messages generated by each algorithm are. Simple messages are shorter and are serialized faster, while complex messages are longer and serialization is

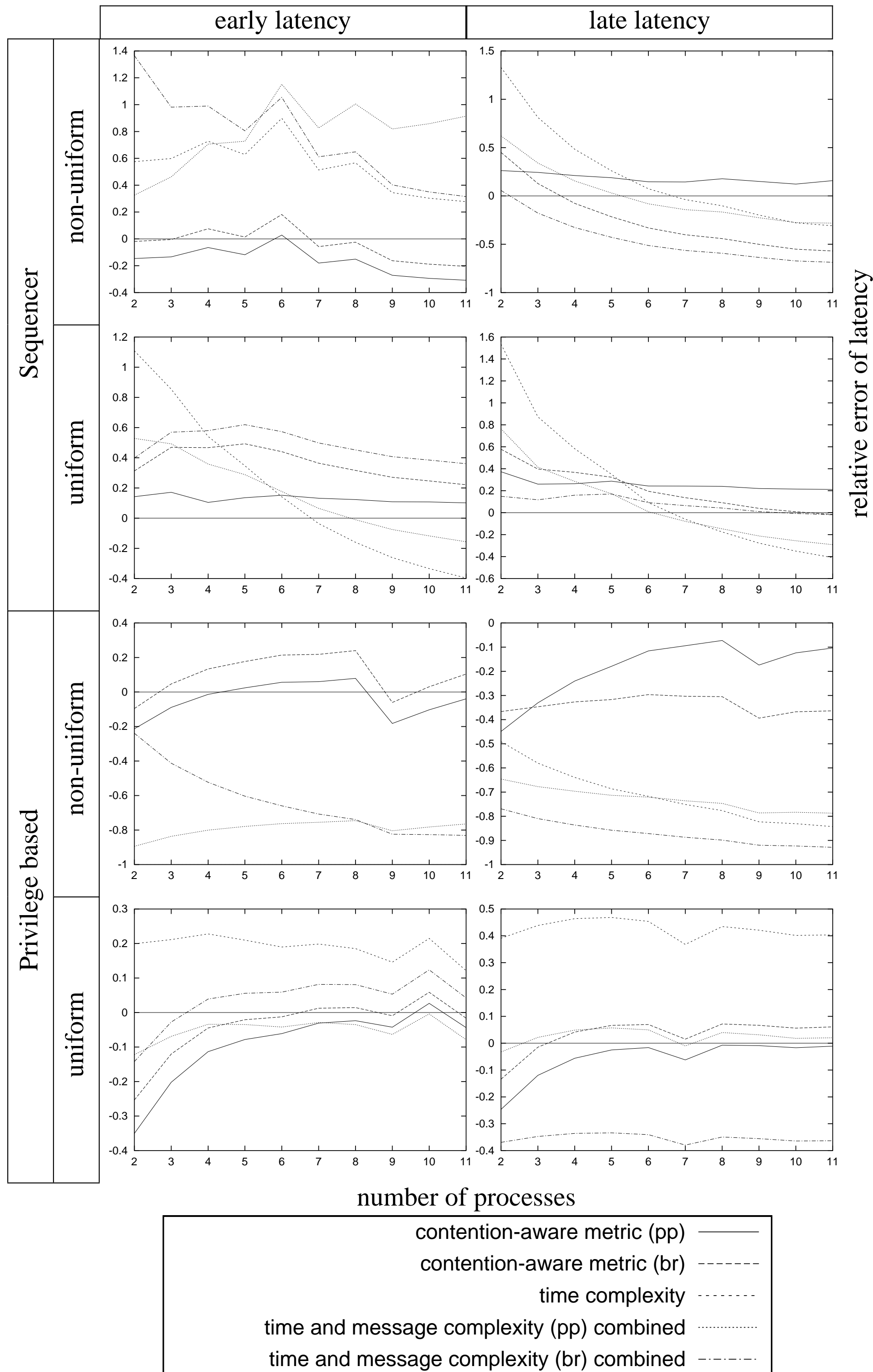


Figure 5.13: Relative error of latency for a variety of metrics (1).

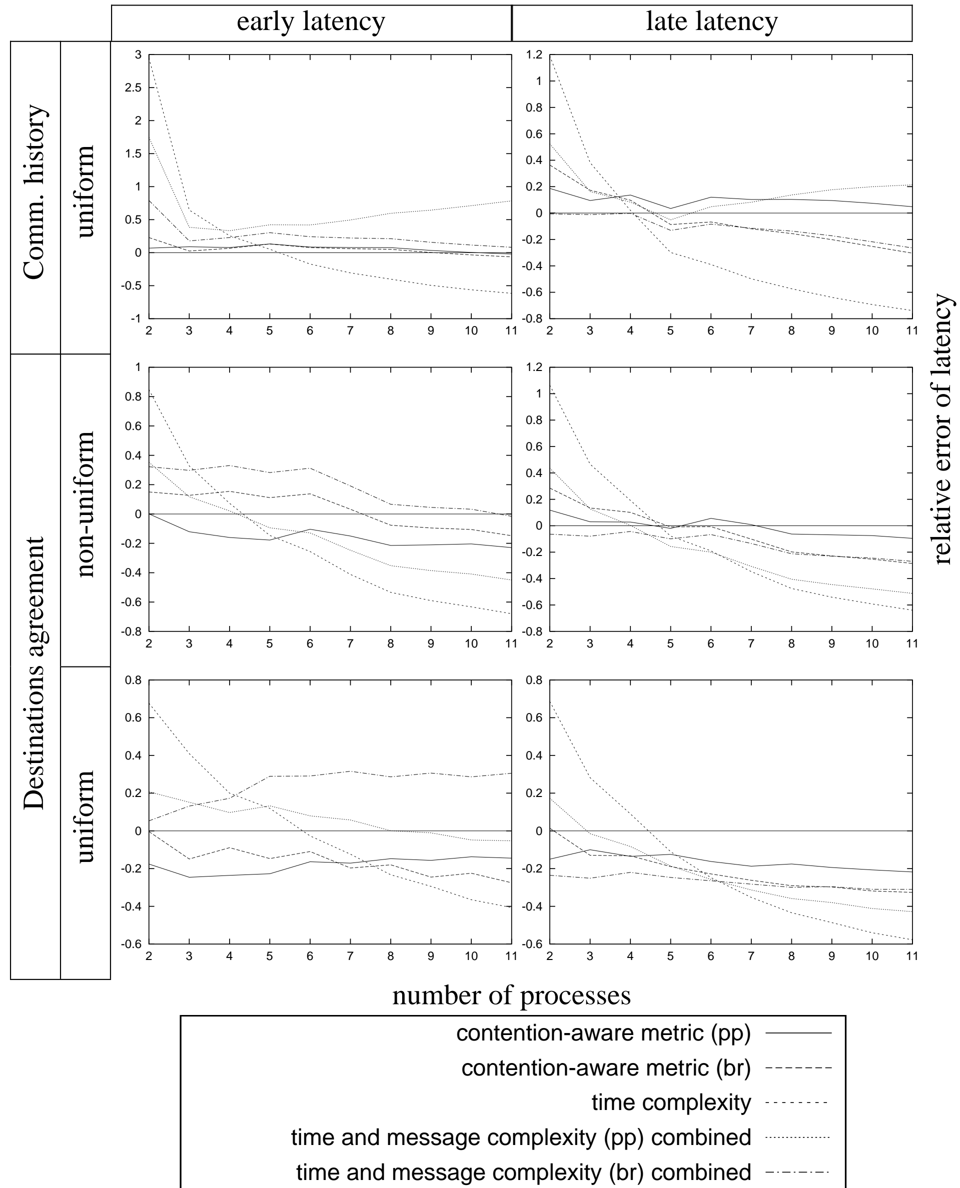


Figure 5.14: Relative error of latency for a variety of metrics (2).

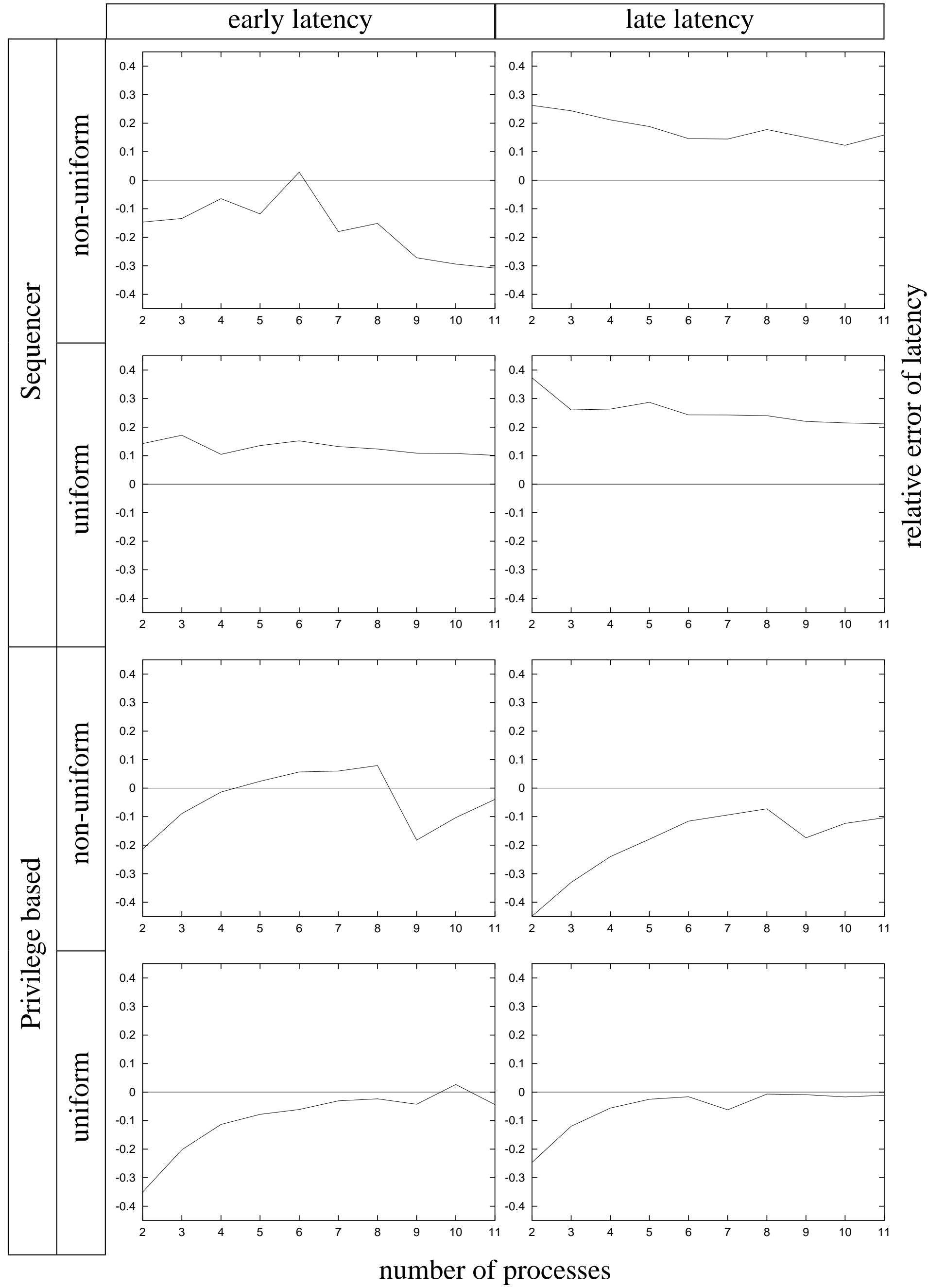


Figure 5.15: Relative error of latency for the contention-aware metric in model $\mathcal{M}_{\rm pp}(n, \lambda)$ (1).

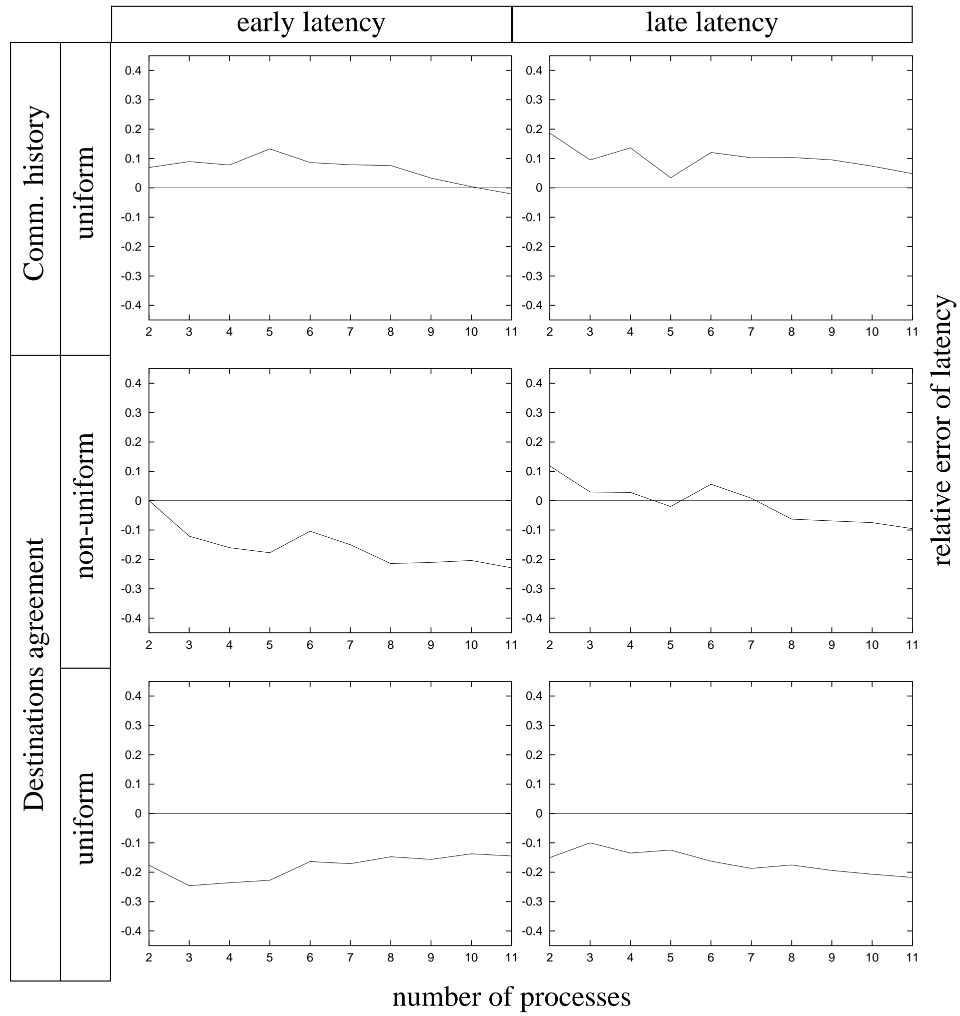


Figure 5.16: Relative error of latency for the contention-aware metric in model $\mathcal{M}_{\mathrm{pp}}(n, \lambda)$ (2).

slower. The metric does not take this difference into account, hence it has a tendency to overestimate the latency of algorithms with simple messages and vice versa. The sequencer and the communication history algorithms generate simple messages (the timestamped atomic broadcast message, or just a timestamp); all the corresponding graphs (except early latency, non-uniform sequencer algorithm) show that latency is underestimated. On the other hand, the destinations agreement algorithms and the uniform privilege-based algorithm use complex messages, and their latencies are underestimated.

- One can also see that the (signed) relative error is lower on the graph showing the early latency of an algorithm (leftmost graph in a row) than on the graph showing the late latency (rightmost graph in the same row). This holds for all algorithms except the non-uniform privilege-based algorithm. Hence there seems to be a factor that increases late latency more than early latency, and that the contention-aware metric does not account for.

  This factor is probably the distribution of message transmission times in an Ethernet network: as Fig. 5.17 shows, the distribution is bimodal, with a mode at ≈ 0.2 ms and another at ≈ 0.7 ms, the latter representing about 6% of all messages. The causal chain of messages leading to the last A-deliver event of an atomic broadcast (used in defining late latency) is likely longer than the chain leading to the first A-deliver event (used in defining early latency), hence the fact that the distribution is bimodal increases late latency to a greater extent. The contention-aware metric does not represent this bi-modal distribution: the end-to-end transmission time is constant if no resource in the network model is busy.

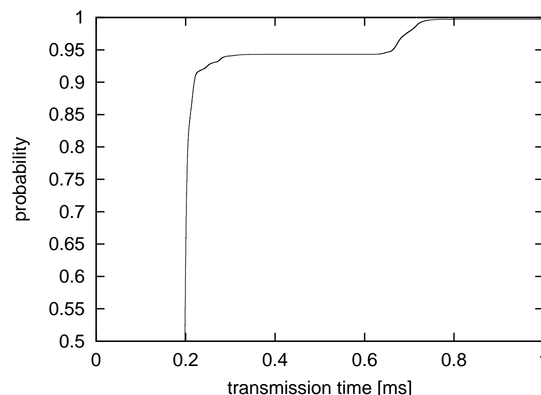


Figure 5.17: Cumulative distribution of the end-to-end transmission time in a lightly loaded Ethernet network.

Chapter 6

Benchmarks for atomic broadcast algorithms

Agreement problems (see Section 2.2) have been extensively studied in various system models, and many protocols solving these problems, with different levels of guarantees, have been published [BMD93, DSU00]. However, these protocols have mostly been analyzed from the point of view of their safety and liveness properties, and little has been done to analyze their *performance*. Moreover, it is difficult to compare the results of performance studies of agreement algorithms as the assumptions of each study are wildly different: there is no agreement on the metrics and the workloads used to analyze algorithms. Also, most papers focus on analyzing failure free runs, thus neglecting the performance aspects of failure handling. In our view, the lack of well defined benchmarks, in both failure free scenarios and scenarios with failure handling, is an obstacle for adopting such protocols in practice.

This chapter defines benchmarks for atomic broadcast. These benchmarks are extensively used in the performance studies in Chapters 7 and 8. The benchmarks include well-defined metrics, workloads and failure scenarios (faultloads). The focus is on defining the metrics and faultloads; in the case of workloads, we preferred simplicity over detailed, realistic models. The reason is that metrics and faultloads are more likely to be relevant for several performance studies, whereas the types of workloads vary a lot from one study to another.

The chapter is structured as follows. We start by describing performance metrics in Section 6.1, followed by workloads in Section 6.2. Faultloads, i.e., the part of the workload that describes how failures and failure suspicions occur, are presented in Section 6.3. Related work is presented in Section 6.4.

6.1 Performance metrics

In this section, we are only concerned with speed metrics (see Section 3.2 for the terminology); reliability and availability metrics are discussed in Section 6.3.

We are looking for metrics defined in terms of the *interface* to atomic broadcast, i.e., the time and frequency of occurrences of A-broadcast and A-deliver events. This ensures that our metrics are valid for all possible atomic broadcast algorithms, which would not be the case if we used details of the implementation.

6.1.1 Latency metrics

We define several responsiveness metrics for atomic broadcast, i.e., metrics related to the time it takes the system to respond to an A-broadcast event requesting the transmission of message m by A-delivering m on processes. Let the A-broadcast(m) event occur at time t_0. and the corresponding A-deliver(m) event on p_i at time t_i, for each $i = 1, \ldots, n$. Each variant of latency, denoted by L and an index, measures the time that elapses from sending the message until it is delivered on some process. *Early latency* is defined as the time until the first delivery:

$$L_{early} \stackrel{\text{def}}{=} \left(\min_{i=1,\ldots,n} t_i \right) - t_0$$

Late latency is defined as the time until the last delivery:

$$L_{late} \stackrel{\text{def}}{=} \left(\max_{i=1,\ldots,n} t_i \right) - t_0$$

Note that this definition works even when processes crash, if we consider that t_i is not defined if process p_i crashes (and thus never delivers m).

A good metric for a system component reflects an aspect of the component's performance that is important for the system using the component. For this reason, we now list systems which include an atomic broadcast algorithm and in which L_{early} and L_{late} are relevant:

- Consider a service replicated using active replication [Sch90]. Clients of this service send their requests to the server replicas using atomic broadcast. Once a request is delivered, the server replica processes the client request, and sends back a reply. The client waits for the first reply, and discards the other ones (identical to the first one). If we assume that the time to service a request is the same on all replicas, and the time to send the response from a server to the client is the same for all servers, then the first response received by the client is the response sent by the server to which the request was delivered first. Thus there is a direct link between the response time of the replicated server and the early latency L_{early}.

- L_{late} is related to the time after which the request takes effect on all replicas of the replicated server.

  L_{late} is important for another reason as well. We are usually interested in the mean latency of an algorithm at a certain workload, in a steady state of the system. As an atomic broadcast algorithm must deliver the message to

all destinations, the system is in a steady state if *all* mean latencies stabilize over time. As L_{late} is the highest of all possible definitions of latency, it is actually enough to check if L_{late} stabilizes over time.

In this thesis, we always use L_{early} and L_{late}. There are other possibilities to define latency metrics whose use can be justified. Two examples follow. *Majority latency* is defined as the time until a majority of all processes deliver the message:

$$L_{maj} \stackrel{\text{def}}{=} t_{(\lceil\frac{n+1}{2}\rceil)} - t_0$$

where $t_{(j)}$ is the j-th smallest element of $\{t_1, \ldots, t_n\}$. *Average latency* is defined as the mean time until delivery, averaged over the set of destination processes:

$$L_{avg} \stackrel{\text{def}}{=} \frac{\sum_{i=1,\ldots,n} t_i}{n} - t_0$$

The following inequalities hold for the latency metrics: $L_{early} \leq L_{maj} \leq L_{late}$ and $L_{early} \leq L_{avg} \leq L_{late}$. Each latency metric is an LB (lower is better) metric.

6.1.2 Maximum throughput

An atomic broadcast algorithm must deliver all messages, hence the rate of sending is equal to the rate of delivery if the system reaches a steady state. We only define one productivity metric: the maximum throughput T_{max}. The metric T_{max} is the highest number of requests per second with which the system is able to reach a steady state, i.e., the late latency L_{late} stabilizes (if L_{late} stabilizes, all latency metrics stabilize, as they are smaller than L_{late}).[1] T_{max} is a HB (higher is better) metric.

We would like to point out that T_{max} might be difficult to measure. One possible reason is that, once under high load, a number of algorithms have only a small marginal cost for each additional atomic broadcast. E.g., the algorithm in [Cri91] uses a token that circulates on a ring to carry all information. With this algorithm, increasing the rate of atomic broadcasts only increases the length of the token message, but does not generate any additional messages. Another possible difficulty is that a lot of algorithms (e.g., [Cri91]) require the set of senders to be a part of the set of destinations (*closed group* algorithms; see [DSU00]). With such algorithms, the contributions of (1) the generation of A-broadcast events and (2) the atomic broadcast algorithm to the overall load on CPUs might be comparable. A value for T_{max} measured with such a algorithm is tied more to the scheduling policies of the operating system than to any objective characteristic of the algorithm itself.

[1] Another possibility would be to define a maximum acceptable level of latency and find the corresponding throughput.

6.1.3 Notes

The metrics defined in this section are not specific to atomic broadcast: they can be used with any broadcast primitive that is not allowed to lose messages, e.g., reliable broadcast (see Section 2.2).[2]

We defined responsiveness and productivity metrics, but no utilization metrics. The reason is that (1) such metrics are not specific to atomic broadcast and that (2) the performance studies in the following chapters do not consider utilization metrics.

6.2 Workloads

In all the workloads considered, processes send atomic broadcasts, each to the same destination group $p_1, \ldots, p_n$. The workloads specify how the A-broadcast events are generated. The generation of A-broadcast events has three aspects: (1) the set of senders, (2) the number of broadcasts per second and (3) the distribution of these broadcasts.

Set of senders. We can include all destination processes or a subset thereof in the set of senders. Also, if the algorithm accepts broadcasts from *external* processes, processes which are not destinations (*open group* algorithms [DSU00]) we might include those as well. We now list a few simple choices for the set of senders:

All destination processes The workload on the system is symmetric. The case studies in this thesis use this workload.

External processes The workload on the system is symmetric. Compared to Case 1, this choice is less general: it only applies to open group algorithms.

Two destination processes The workload is asymmetric. Certain algorithms can take advantage of this: e.g., *privilege based* algorithms [DSU00], in which the privilege to send is passed to processes that send a lot of messages [CMA97].

One destination process In this extreme case with asymmetric load, no ordering needs to be done. This choice measures how well an algorithm can take advantage of this fact.

Throughput. The throughput T is the number of A-deliver events per second. In a steady state of the system, it is equal to the number of A-broadcast events per destination process per second, as atomic broadcast may not lose messages (unless processes crash). For this reason, T is a parameter rather than a metric in our case studies.

[2] With lossy broadcast primitives, metrics for the loss rate should be defined in addition to these metrics.

Distribution of broadcasts. In this thesis, we assume that all processes generate atomic broadcasts independently, at the same rate. As for the distribution of broadcasts sent by a given process, the A-broadcast events come from a Poisson (stochastic) process, i.e., the time between two events on the same process follows an exponential distribution.[3] This is the simplest possible choice. Nevertheless, it models certain kinds of systems rather well: e.g., systems in which a lot of clients delegate the execution of their requests to one server of a group of servers, set up in a load balancing configuration. The fact that each server aggregates requests from a lot of sources results in exponential inter-arrival times. A cluster of web servers may exhibit such a behavior.

Notes. The workloads defined in this section can be used with any broadcast primitive that is not allowed to lose messages, like reliable broadcast. However, some choices for the sets of senders (one, two and all destination processes) were only presented because they allow to test how efficiently messages are ordered, and are thus more interesting when used with atomic broadcast.

6.3 Faultloads

The faultload is the part of the workload that describes failure-related events that occur during an experiment [MKA+01]. We concentrate on (1) crash failures of processes, and (2) the behavior of unreliable failure detectors. We first clarify our assumptions about the system in which the algorithms run, and then present a number of representative faultloads. Finally, we present an abstract model of the behavior of failure detectors.

6.3.1 Assumptions about the system

We choose faultloads based on the following assumptions about the system:

- Processes only fail by crashing, and the network is reliable. Process failures are independent. This is the failure model that all algorithms in this thesis (and a major part of all published atomic broadcast algorithms) use.
- Process crashes are rare, and process recovery is slow (or processes do not recover at all): both the time between crashes and the time to repair are much greater than the latency of the atomic broadcast algorithms or the time needed to reach a steady state after a crash.

  We need these assumptions to define the scope of our studies: the focus is the steady-state performance of the algorithms, as well as the transient state performance after a crash. Issues that happen on a greater timescale are out

[3] The only exception is when we investigate algorithms at a very low throughput, such that any two subsequent atomic broadcasts are independent. In this case, only one process generates A-Broadcast events, and the time between two events is a constant higher than the late latency L_{late}.

of scope: this includes (1) the details of process recovery, (2) dependability metrics like MTTF (mean time to failure) and (3) scenarios in which multiple process crashes occur within a short time. Note that studying these issues would require a number of additional assumptions: e.g., defining a cost parameter for process recovery or characterizing how often crash failures occur.

- Failure detectors are unreliable (see Section 2.1.3.A). We study (1) how fast they detect failures and (2) how often and for what duration they wrongly suspect correct processes. Note that whereas we assume that process crashes are rare, (wrong) failure suspicions may occur frequently, depending on the tuning of the failure detectors.

6.3.2 Steady state of the system

We define a number of faultloads for a system in its steady state. Steady state is reached a sufficiently long time after the start of the system or after any crashes. With these faultloads, the performance metrics (e.g., latency) are averaged over a sufficiently long observation period.

We distinguish three faultloads, based on whether crashes and wrong suspicions (failure detectors suspecting correct processes) occur:

- **normal-steady:** Neither crashes nor wrong suspicions occur in the experiment.

- **crash-steady:** One or several crashes occur before the experiment. The parameter of this faultload is the set of crashed processes. As we assume that the crashes happened a long time ago, all failure detectors in the system permanently suspect all crashed processes at this point. No wrong suspicions occur.

- **suspicion-steady:** No crashes occur, but failure detectors generate wrong suspicions. Wrong suspicions generally have a negative impact on performance. The parameters of this faultload describe how often wrong suspicions occur and how long they last. These parameters are discussed in detail in Section 6.3.4.

It would be meaningful to combine the crash-steady and suspicion-steady faultloads, to have both crashes and wrong suspicions. In the performance studies of the following chapters, we omitted this case, for we wanted to observe the effects of crashes and wrong suspicions independently.

6.3.3 Transient state after a crash

In this faultload, we force a crash after the system reached a steady state. After the crash, we can expect a halt or a significant slowdown of the system for a short

period. We would like to capture how the responsiveness metrics (different kinds of latency) change in atomic broadcasts directly affected by the crash. Our faultload definition represents the simplest possible choice: we determine the latency of an atomic broadcast sent at the moment of the crash (by a process other than the crashing process). Of course, the latency of this atomic broadcast may depend on the choice for the sender and the crashing process. In order to reduce the number of parameters, we consider the worst case, i.e., the choice that increases latency the most.

The precise definition for the faultload is the following:

- **crash-transient:** Consider that a process p crashes at time t (no other crashes nor wrong suspicions occur). Let process q ($p \neq q$) execute *A-broadcast*(m) at t. Let $L(p, q)$ be the mean latency of broadcasting m, averaged over a lot of executions. Then $L_{crash} \stackrel{\text{def}}{=} \max_{p,q \in P} L(p, q)$, i.e., we consider the crash that increases the latency most.

The parameter of this faultload describes how fast failure detectors detect the crash (discussed in Section 6.3.4). In fact, we introduce this faultload in order to be able to study this aspect of failure detectors.

We could combine the crash-transient faultload with both the crash-steady and suspicion-steady faultloads, to include other crashes and/or wrong suspicions. In the performance studies of the following chapters, we omitted these cases, for we wanted to observe the effects of (1) the recent crash, (2) old crashes and (3) wrong suspicions independently. Another reason is that we expect the effect of wrong suspicions on latency to be secondary with respect to the effect of the recent crash: wrong suspicions usually happen on a larger timescale.

6.3.4 Modeling failure detectors

One approach to examine the behavior of a failure detector is implementing it and using the implementation in the experiments. However, this approach would restrict the generality of our performance studies: another choice for the algorithm would likely give different results. Also, often it is not justified to model the failure detector in so much detail, as other components of the system, like the execution environment, might be modeled much more coarsely. Yet another reason is that it is difficult to control an implementation such that it exhibits the whole range of possible behaviors of a failure detector: usually, a given implementation in a given system restricts the set of possible behaviors to a great extent.

To answer these concerns, we present a more abstract model of failure detectors, using the notion of quality of service (QoS) introduced in [CTA02]. This model can be used in simulations and in measurements as well (in a fault injection experiment).

The authors of [CTA02] consider the failure detector at a process q that monitors another process p, and identify the following three primary QoS metrics (see Figures 6.1 and 6.2):

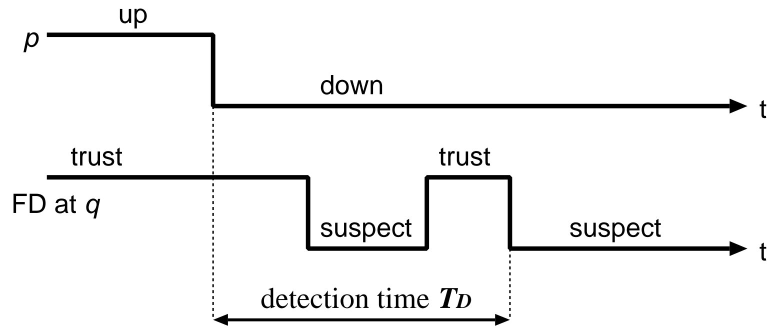


Figure 6.1: Quality of service metric related to the completeness of failure detectors. Process q monitors process p.

Detection time T_D**:** The time that elapses from p's crash to the time when q starts suspecting p permanently.

Mistake recurrence time T_{MR}**:** The time between two consecutive mistakes (q wrongly suspecting p), given that p did not crash.

Mistake duration T_M**:** The time it takes a failure detector component to correct a mistake, i.e., to trust p again (given that p did not crash).

Not all of these metrics are equally important in each of our faultloads (see Section 6.3). In the *normal-steady* faultload, the metrics are not relevant. The same holds in the *crash-steady* faultload, because we observe the system a sufficiently long time after all crashes, long enough to have all failure detectors to suspect the crashed processes permanently. In the *suspicion-steady* faultload, no crash occurs, hence the latency of atomic broadcast only depends on T_{MR} and T_M. In the *crash-transient* faultload, no wrong suspicions occur, hence T_D is the relevant metric.

In [CTA02], the QoS metrics are random variables, defined on a pair of processes. In our system, where n processes monitor each other, we have thus $n(n-1)$

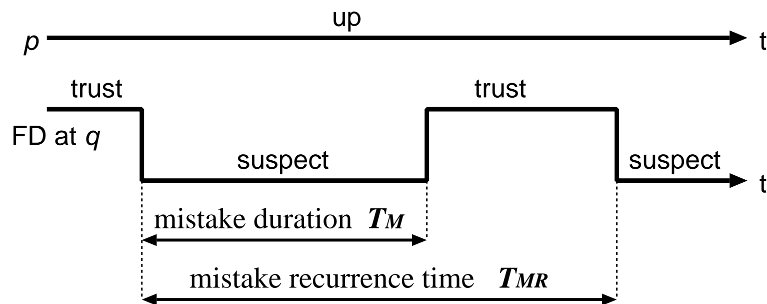


Figure 6.2: Quality of service metrics related to the accuracy of failure detectors. Process q monitors process p.

failure detectors in the sense of [CTA02], each characterized with three random variables. In order to have an executable model for the failure detectors, we have to define (1) how these random variables depend on each other, and (2) how the distribution of each random variable can be characterized. To keep our model simple, we assume that all failure detector modules are independent and the tuples of their random variables are identically distributed. Moreover, note that we do not need to model how T_{MR} and T_M depend on T_D, as the two former are only relevant in the *suspicion-steady* faultload, whereas T_D is only relevant in the *crash-transient* faultload. In the performance studies of the following chapters, we considered various settings for T_D, and various settings for combinations of T_{MR} and T_M. As for the distributions of the metrics, we took the simplest possible choices: T_D is a constant, and both T_{MR} and T_M are exponentially distributed with (different) constant parameters.

Note that these modeling choices are not realistic: suspicions from different failure detectors are probably correlated in a real system, as wrong failure suspicions might be caused by an overload situation that affects several or even all processes. Our choices only represents a starting point, as we are not aware of any previous work we could build on (apart from [CTA02] that makes similar assumptions). We will refine our models as we gain more experience.

Finally, note that this abstract model for failure detectors neglects that failure detectors and their messages put a load on system components. This simplification is justified in a variety of systems, in which a rather good QoS can be achieved with failure detectors that send messages infrequently.

6.4 Related work

In this section, we review a number of papers that analyze the performance of atomic broadcast algorithms, or other related algorithms that tolerate crash failures: consensus [SDS01, CUBS02, HUSK02] (defined in Section 2.2.1) and terminating reliable broadcast [DCS97].[4] We do not consider (1) papers using only complexity metrics on an isolated atomic broadcast execution (a lot of papers describing new atomic broadcast algorithms have a section with such a simplistic analysis); or (2) studying atomic broadcast as a component of a larger system (e.g, a replicated database [Wie02]).

Metrics. A lot of papers consider latency metrics. The most popular metric is the late latency L_{late} [MSS95, GMS91, CdBM94, CMA97], also called termination time, broadcast delivery time or broadcast stability time; the latter two terms are used for algorithms that provide two delivery events per broadcast, the

[4] Terminating reliable broadcast is like reliable broadcast (see Section 2.2.2). The difference is that the destinations are always delivered something if the sender crashes: either the message sent, or a special message indicating the crash of the sender. In a system with Byzantine failures, such a primitive is usually referred to as Byzantine agreement or the Byzantine generals problem.

first signaling non-uniform delivery and the second uniform delivery [CdBM94, CMA97]. Papers often implicitly mean L_{late} by latency (e.g., [BC94]). [SDS01] uses an analogous latency definition for consensus. The early latency L_{early} is used in [PSUC02a], and a latency metric analogous to L_{early} is used in [CUBS02, HUSK02]. L_{avg} is used in [May92]. [FvR97, RPL+02] define the latencies of more complex message exchanges that occur in their workloads; in fact, these latencies are inversely proportional to the maximal throughput T_{max} (see the next paragraph).

[CSG+00, CBG01, FvR97, BC94] consider maximal throughput T_{max} as a metric. All the algorithms analyzed include some (sometimes implicit) flow control mechanism (see also the workloads paragraph). If we consider that *A-broadcast* events can be generated at any rate (i.e., generating them is not subject to flow control) our definition of T_{max} applies to these studies: if the throughput is higher than what flow control allows, *A-broadcast* events end up in ever growing queues and thus L_{late} never stabilizes.

Some papers also define utilization metrics: channel utilization [MSS95] and CPU load distribution [CdBM94, CMA97], the maximal size of the algorithms' buffers [CM84, CdBM94, CMA97] and the number of messages per broadcast [CM84, GMS91, CdBM94, CMA97].

Workloads. The most frequently used workload is the isolated atomic broadcast [May92, GMS91, DCS97] or consensus [SDS01, CUBS02, HUSK02], which is a special case of our workloads, corresponding to a very low setting for the throughput. Results obtained with this workload are only applicable to (very) lightly loaded systems.

[May92] also defines two other workloads different from ours: one at very low throughput, with two processes sending at the same time (in order to exercise the ordering mechanism in the algorithm); and another in which one process broadcasts at regular intervals.

Workloads in other papers have *A-broadcast* events generated by a Poisson process, just like our workloads do [CM84, CdBM94, CMA97, MSS95, PSUC02a]. The set of senders is the set of all destinations in each paper.

Poisson workloads assume a nearly constant rate of broadcast arrivals on each sender process. A lot of real systems, however, have broadcasts arriving in bursts. In the networking literature, self-similar models [LTWW94] are often used to characterize such arrival patterns. We might need to define such workloads in the future, as there are atomic broadcast algorithms (e.g., On-demand in [CMA97]) optimized for bursty arrival patterns.

Another common workload [BC94, FvR97, RPL+02, CSG+00, CBG01] is one in which processes send atomic broadcasts as often as the flow control mechanism of the algorithm allows. All the algorithms in these papers can be understood as algorithms having a built-in flow control mechanism: in [BC94], flow control mechanisms are described as such; in [CSG+00, CBG01], the polling performed

by the wireless access point plays this role; in [FvR97, RPL$^+$02], only a fixed number of outstanding atomic broadcasts are allowed per sending process. In all the papers, the set of senders is all destination processes, except for [FvR97], which also considers one destination process as sender (and needs acknowledgment messages from all other processes to implement flow control), and [BC94] where the set of senders is unclear.

Faultloads. Most papers only consider failure free executions (our normal-steady faultload), which only gives a partial and incomplete understanding of the behavior of the algorithms. We only note the exceptions here, starting with the papers that have faultloads similar to ours, concerned with process crashes and their detection.

The permanent effects of crashes are studied relatively often: [CUBS02] and [HUSK02] use the crash-steady faultload.

The transient effects of a crash are studied in [RPL$^+$02, SDS01], but the faultload is different from our crash-transient faultload. [SDS01] assumes that the crash occurs at the worst possible moment during an atomic broadcast, in order to obtain the worst case latency in the case of a crash. In contrast to the crash-transient faultload, this faultload requires a detailed knowledge of the execution. This knowledge is only available if one considers very simple workloads (isolated executions of consensus in [SDS01]) in an analytical or simulation model; in particular, this faultload is not suited for measurements. The other paper [RPL$^+$02] measures the latency of the group membership service used by the algorithm to tolerate crash failures.[5] This way of considering the transient effects of a crash is less general compared to our crash-transient faultload, as it is stated in terms of an implementation detail of the atomic broadcast algorithm.

The effect of wrong failure suspicions on atomic broadcast algorithm is studied, to our knowledge, only in our previous work [CUBS02, PSUC02a]. The former paper uses the suspicion-steady faultload, while the latter uses a faultload whose parameter is the timeout value used by a particular implementation.

Only three papers investigate failure detectors in detail. [SDS01, PSUC02a] uses particular implementations of failure detectors rather than an abstract model. [SDS01] considers several implementations and compares their effects. Our previous work [CUBS02] performs simulations with the abstract model of Section 6.3.4 and measurements with an implementation (and contrasts the results).

We assume reliable communication in all our performance studies. A number of papers, however, consider message losses. [CM84, MSS95, CSG$^+$00, CBG01, DCS97] assign a fixed probability for losing each message, whereas [CdBM94, CMA97] consider one message loss in the messages generated by each atomic broadcast.

[DCS97, CSG$^+$00, CBG01] are also concerned with dependability metrics, which are out of scope for our studies. They assign a fixed probability for losing each message, and, in the case of [DCS97], also a fixed rate for the crash of each

[5] Certain kinds of Byzantine failures are also injected.

process. Over time, this may lead to a violation of the algorithms' assumptions about the system (number of failures / subsequent omissions); the papers derive the probability that the algorithm is still working after a certain time.

Isolating the cost of ordering. Conceptually, atomic broadcast consists of reliable broadcast and an ordering guarantee (see Section 2.2.3). Some papers attempt separating the costs of ordering from the cost of a reliable broadcast. [RPL+02] analyzes both a reliable broadcast algorithm and its extension to an atomic broadcast algorithm, and contrasts the results. [May92] also assumes an architecture in which atomic broadcast starts with a reliable broadcast and exchanges some messages only concerned with ordering. The time for which a message is delayed from the moment of its receipt is summed over all destinations and all messages (a similar definition to the average latency metric L_{avg}). The limitation of this approach is that not all atomic broadcast algorithms start with a reliable broadcast or an equivalent exchange of messages. A whole class of algorithms (called privilege based in [DSU00]) delays the sending of messages rather than their delivery in order to obtain total ordering. Nevertheless, we consider isolating the cost of ordering an important step towards the understanding the performance trade-offs of atomic broadcast algorithms.

Chapter 7

Comparison of failure detectors and group membership: performance study of two atomic broadcast algorithms

Unreliable failure detectors *vs.* group membership. In this chapter, we compare two (uniform) atomic broadcast algorithms, the one based on *unreliable failure detectors* and the other on a *group membership service*. Both services provide processes with estimates about the set of crashed processes in the system.[1] The main difference is that failure detectors provide inconsistent information about failures, whereas a group membership service provides consistent information. While several atomic broadcast algorithms based on unreliable failure detectors have been described in the literature, to the best of our knowledge, all existing group communication systems provide an atomic broadcast algorithm based on group membership (see [CKV01] for a survey). So indirectly our study compares two classes of techniques, one widely used in implementations (based on group membership), and the other (based on failure detectors) not (yet) adopted in practice.

The two algorithms. The algorithm that uses unreliable failure detectors is the Chandra-Toueg atomic broadcast algorithm [CT96] with one of the consensus algorithms described in the same paper, which can tolerate $f < n/2$ crash failures, and requires the failure detector $\Diamond\mathcal{S}$. As for an algorithm using group membership, we chose an algorithm that implements total order with a mechanism close to the failure detector based algorithm, i.e., a sequencer based algorithm (which also tolerates $f < n/2$ crash failures). Both algorithms were optimized (1) for failure and suspicion free runs (rather than runs with failures and suspicions), (2) to minimize

[1] Beside masking failures, a group membership service has other uses. This issue is discussed in Section 7.4.

latency under low load (rather than minimize the number of messages), and (3) to tolerate high load (rather than minimize latency at moderate load).

We chose these algorithms because they are well-known and easily comparable: they offer the same guarantees in the same model. Moreover, they behave similarly if neither failures nor failure suspicions occur (in fact, they generate the same exchange of messages given the same arrival pattern). This allows us to focus our study on the differences in handling failures and suspicions.

Elements of the performance study. The two algorithms are evaluated using simulation. We model message exchange by taking into account contention on the network and the hosts, using the metrics described in Chapter 5. We model failure detectors (including the ones underlying group membership) in an abstract way, using the quality of service (QoS) metrics proposed by Chen et al. [CTA02] (see Section 6.3.4). We study the atomic broadcast algorithms in several benchmarks defined in Chapter 6, including scenarios with failures and suspicions: we evaluate the steady state latency in (1) runs with neither crashes nor suspicions, (2) runs with crashes and (3) runs with no crashes in which correct processes are wrongly suspected to have crashed, as well as (4) the transient latency after a crash.

The results. We show that the two algorithms have the same performance in runs with neither crashes nor suspicions, and that the group membership based algorithm has an advantage in terms of performance and resiliency a long time after crashes occur. In the other scenarios, involving wrong suspicions of correct processes and the transient behavior after crashes, the failure detector based algorithm offers better performance. We discuss the implications of our results to the design of fault tolerant distributed systems.

Structure. The rest of the chapter is structured as follows. We introduce the algorithms and discuss their expected performance in Section 7.1. Section 7.2 describes all important elements of our performance study. Our results are presented in Section 7.3, and the chapter concludes with a discussion in Section 7.4.

7.1 Algorithms

This section introduces the two atomic broadcast algorithms and the group membership algorithm. Then we discuss the expected performance of the two atomic broadcast algorithms.

7.1.1 Chandra-Toueg uniform atomic broadcast algorithm

The Chandra-Toueg uniform atomic broadcast algorithm (with one of the consensus algorithms described in [CT96]; see below) uses failure detectors directly

[CT96] (see Sections 2.2.3 and 2.1.3.A for definitions). We shall refer to it as the FD atomic broadcast algorithm, or simply as the *FD algorithm*. A process executes A-broadcast by sending a message to all processes.[2] When a process receives such a message, it buffers it until the delivery order is decided. The delivery order is decided by a sequence of consensus numbered 1, 2, ... The initial value and the decision of each consensus is a *set of message identifiers*. Let $msg(k)$ be the set of message IDs decided by consensus $\#k$. The messages denoted by $msg(k)$ are A-delivered before the messages denoted by $msg(k+1)$, and the messages denoted by $msg(k)$ are A-delivered according to a deterministic function, e.g., according to the order of their IDs.

A detailed pseudocode description of the algorithm is given in Section A.2.7, in the Appendix.

Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm. For solving consensus (defined in Section 2.2.1) we use the Chandra-Toueg $\diamond\mathcal{S}$ algorithm [CT96].[3] The algorithm tolerates $f < n/2$ crash failures. It is based on the rotating coordinator paradigm: each process executes a sequence of asynchronous rounds (i.e., not all processes necessarily execute the same round at a given time t), and in each round a process takes the role of *coordinator* (p_i is coordinator for rounds $kn + i$). The role of the coordinator is to impose a decision value on all processes. If it succeeds, the consensus algorithm terminates. It may fail if some processes *suspect* the coordinator to have crashed (whether the coordinator really crashed or not). In this case, a new round is started.

The details of the execution are not necessary for understanding the rest of the chapter. A detailed pseudocode description is given in Section B.2, in the Appendix.

Example run of the FD algorithm. Figure 7.1 illustrates an execution of the FD atomic broadcast algorithm in which one single message m is A-broadcast and neither crashes nor suspicions occur. At first, m is sent to all processes. Upon receipt, the consensus algorithm starts. The coordinator sends its proposal to all other processes. Each process acknowledges this message. Upon receiving acks from a majority of processes (including itself), the coordinator decides its own proposal and sends the decision (using reliable broadcast) to all other processes. The other processes decide upon receiving the decision message.

7.1.2 Fixed sequencer uniform atomic broadcast algorithm

The second uniform atomic broadcast algorithm is based on a fixed sequencer [BSS91]. It uses a group membership service for reconfiguration in case of a

[2] This message is sent using reliable broadcast. We use an efficient algorithm inspired by [FP01] that requires one broadcast message if the sender is not suspected. The algorithm is described in Section B.1, in the Appendix.

[3] Actually, we included some easy optimizations in the algorithm, described in Section 8.1.2.

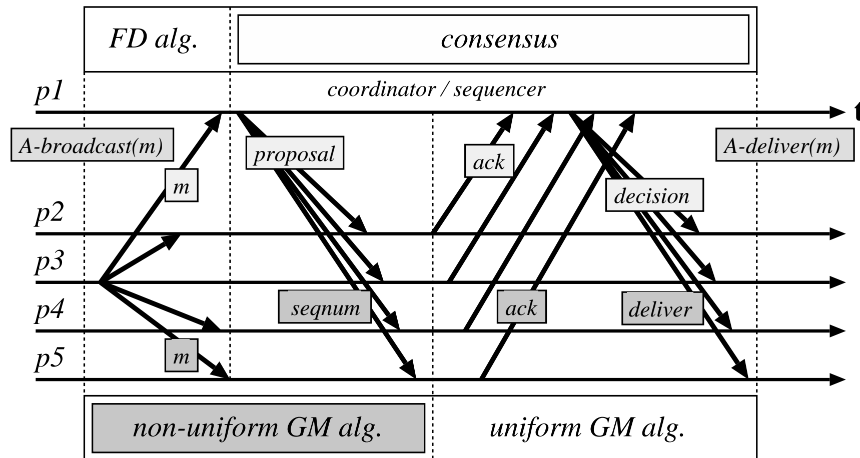


Figure 7.1: Example run of the atomic broadcast algorithms. Labels on the top/bottom refer to the FD/GM algorithm, respectively.

crash. We shall refer to it as the GM atomic broadcast algorithm, or simply as the *GM algorithm*. We describe here the *uniform* version of the algorithm.

In the GM algorithm, one of the processes takes the role of *sequencer*. When a process A-broadcasts a message m, it first broadcasts it to all. Upon reception, the sequencer (1) assigns a sequence number to m, and (2) broadcasts the sequence number to all. When non-sequencer processes have received m and its sequence number, they send an acknowledgment to the sequencer.[4] The sequencer waits for acks from a majority of processes, then delivers m and sends a message indicating that m can be A-delivered. The other processes A-deliver m when they receive this message. The execution is shown in Fig. 7.1. A pseudocode description of the algorithm is given in Section A.2.2, in the Appendix.

Note that the messages denoted *seqnum*, *ack* and *deliver* can carry several sequence numbers.[5] This is essential for achieving good performance under high load. Note that the FD algorithm has a similar "aggregation" mechanism: one execution of the consensus algorithm can decide on the delivery order of several messages.

When the sequencer crashes, processes need to agree on a new sequencer. This is why we need a group membership service: it provides a consistent *view* of the group to all its members, i.e., a list of the processes which have not crashed (informally speaking). The sequencer is the first process in the current view. The group membership algorithm described below can tolerate $f < n/2$ crash failures (more in some runs) and requires the failure detector $\diamond\mathcal{S}$.

[4] Figure 7.1 shows that the acknowledgments and subsequent messages are not needed in the non-uniform version of the algorithm. We come back to the issue of uniformity in Section 7.4.

[5] This aspect does not appear in the pseudocode.

7.1.3 Group membership algorithm

A group membership service (see Section 2.2.4) maintains the *view* of a group, i.e., the list of correct processes of the group. The current view[6] might change because processes in the group might crash or exclude themselves, and processes outside the group might join. The group membership service guarantees that processes see the same sequence of views (except for processes which are excluded from the group; they miss all views after their exclusion until they join again). In addition to maintaining the view, our group membership service ensures *View Synchrony* and *Same View Delivery*: correct and not suspected processes deliver the same set of messages in each view, and all deliveries of a message m take place in the same view.

The group membership algorithm [MS95] uses failure detectors to start view changes, and relies on consensus to agree on the next view. This is done as follows. A process that suspects another process starts a view change by sending a "view change" message to all members of the current view. As soon as a process learns about a view change, it sends its unstable messages to all others.[7] When a process has received the unstable messages from all processes it does not suspect, say P, it computes the union U of the unstable messages received, and starts consensus with the pair (P, U) as its initial value. Let (P', U') be the decision of the consensus. Once a process decides, it delivers all messages from U' not yet delivered (all processes deliver the messages in the same deterministic order) and installs P' as the next view. The protocol for joins and explicit leaves is very similar.

State transfer. When a process joins a group, its state needs to be synchronized with the other members of the group. What "state" and "synchronizing" exactly mean is application dependent. We only need to define these terms in a limited context: in our study, the only processes that ever join are correct processes which have been wrongly excluded from the group. Consequently, the state of such a process p is mostly up-to-date. For this reason, it is feasible to update the state of p the following way: when p rejoins, it asks some process for the messages it has missed since it was excluded. Process p delivers these messages, and then starts to participate in the view it has joined. Note that this only works because our atomic broadcast algorithm is uniform: with non-uniform atomic broadcast, the excluded process might have delivered messages never seen by the others, thus having an inconsistent state. In this case, state transfer would be more complicated.

[6]There is only one current view, since we consider a *non-partitionable* or *primary partition* group membership service.

[7]Message m is *stable* for process p when p knows that m has been received by all other processes in the current view. The goal of this message exchange is to ensure that all messages sent in the current view are stable.

7.1.4 Expected performance

We now discuss, from a qualitative point of view, the expected relative performance of the two atomic broadcast algorithms (FD algorithm and GM algorithm).

Figure 7.1 shows executions with neither crashes nor suspicions. In terms of the pattern of message exchanges, the two algorithms are identical: only the content of messages differ. Therefore we expect the same performance from the two algorithms in failure free and suspicion-free runs.

Let us now investigate how the algorithms slow down when a process crashes. There are two major differences. The first is that the GM algorithm reacts to the crash of *every* process, while the FD algorithm reacts only to the crash of p_1, the first coordinator. The other difference is that the GM algorithm takes a longer time to re-start delivering atomic broadcast messages after a crash. This is true even if we compare the GM algorithm to the worst case for the FD algorithm, i.e., when the first coordinator p_1 fails. The FD algorithm needs to execute Round 2 of the consensus algorithm. This additional cost is comparable to the cost of an execution with no crashes (3 communication steps, 1 multicast and about $2n$ unicast messages). On the other hand, the GM algorithm initiates an expensive view change (5 communication steps, about n multicast and n unicast messages). Hence we expect that if the failure detectors detect the crash in the same time by the two algorithms, the FD algorithm performs better.

Consider now the case when a correct process is wrongly suspected. The algorithms react to a wrong suspicion the same way as they react to a real crash. Therefore we expect that if the failure detectors generate wrong suspicions at the same rate, the FD algorithm will suffer less performance penalty.

7.2 Elements of our performance study

7.2.1 Performance metrics and workloads

Our main performance metric is the *early latency* of atomic broadcast, i.e., the time that elapses from the broadcast to the first delivery (see Section 6.1). Early latency is simply referred to as latency and is denoted by L. In our study, we compute the mean for L over a lot of messages (thousands) and several independent executions.

Latency is always measured under a certain workload. We chose a symmetric workload where (1) the set of senders is all destination processes and (2) the A-broadcast events follow Poisson arrival (see Section 6.2). The workload is characterized by the throughput T, i.e., the overall rate of atomic broadcast messages.

In general, we determine how the latency L depends on the throughput T. Other parameters that influence latency are the choice for the algorithm A and the number of processes n. We also determine the maximal throughput T_{max}.

7.2.2 Faultloads

We evaluate the latency of the atomic broadcast algorithms using a variety of faultloads from Section 6.3. The steady state latency is determined with the normal-steady, crash-steady and suspicion-steady faultloads. These three faultloads involve neither crashes nor suspicions, crashes, and failure detectors wrongly suspecting correct processes, respectively. The transient latency after a crash is determined with the crash-transient faultload.

Failure detectors are described with an abstract model which was presented in Section 6.3.4. Wrong suspicions are characterized with the two parameters mistake recurrence time T_{MR} and mistake duration T_M. We study how latency depends on these parameters with the suspicion-steady faultload, our only faultload with wrong suspicions. The speed of detecting crash failures is characterized with the detection time parameter T_D, which is a parameter of the crash-transient faultload.

7.2.3 Modeling the execution environment

Our approach to performance evaluation is simulation, which allowed for more general results as would have been feasible to obtain with measurements in a real system (we can use a parameter in our network model to simulate a variety of different environments). We used the Neko prototyping and simulation framework (see Chapter 4) to conduct our experiments.

We used the model of Section 5.3 for the execution environment. This simple model accounts for resource contention on the network and the hosts. We used the broadcast (rather than the point-to-point) variant of the model for our study. The model has one parameter (λ) that reflects the relative cost of processing on the hosts and the network in the transmission of messages. We used several representative settings for this parameter.

It is worthwhile to describe in detail how crashes appear in our model. If a process p_i crashes at time t, no messages can pass between p_i and CPU_i after t; however, the messages on CPU_i and the attached queues are still sent, even after time t. In real systems, this corresponds to a (software) crash of the application process (operating system process), rather than a (hardware) crash of the host or a kernel panic. We chose to model software crashes because they are more frequent in most systems [Gra86].

7.3 Results

We now present the results for all four faultloads. We obtained results for a variety of representative settings for λ: 0.1, 1 and 10. The settings $\lambda = 0.1$ and 10 correspond to systems where communication generates contention mostly on the network and the hosts, respectively, while 1 is an intermediate setting. For example, in current LANs, the time spent on the CPU is much higher than the time spent on the wire, and thus $\lambda = 10$ is probably the setting closest to reality.

Most graphs show latency vs. throughput. The maximum throughput T_{max} is approximately the highest throughput value, that is, the x coordinate of the right-most point, in all graphs showing the steady-state latency; beyond this throughput, the late latency L_{late} (see Section 6.1) did not stabilize. For easier understanding, we set the time unit of the network simulation model to 1 ms. The 95% confidence interval is shown for each point of the graph. The two algorithms were executed with 3 and 7 processes, to tolerate 1 and 3 crashes, respectively.

Normal-steady faultload (Fig. 7.2). With this faultload, the two algorithms have the same performance. Each curve thus shows the latency of *both* algorithms.

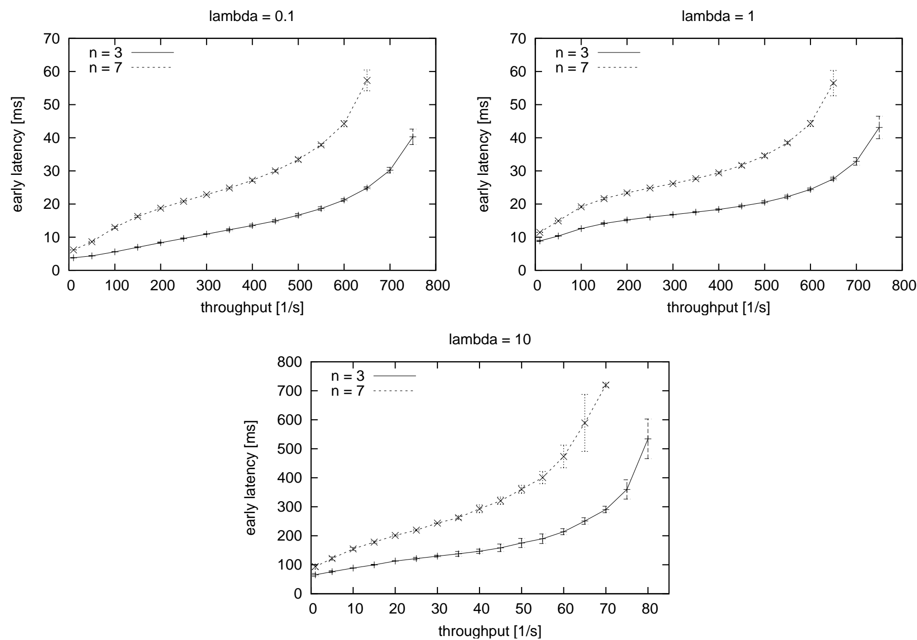


Figure 7.2: Latency vs. throughput with the normal-steady faultload.

Crash-steady faultload (Fig. 7.3). The figure shows that the GM algorithm offers a slightly lower latency at the same throughput, as well as a slightly higher maximum throughput, than the FD algorithm. With both algorithms, the latency decreases as more processes crash. This is due to the fact that the crashed processes do not load the network with messages. The GM algorithm has an additional feature that improves performance: the sequencer waits for fewer acknowledgments,[8] as the group size decreases with the crashes. By comparison, the coordinator in the FD algorithm always waits for the same number of acknowledgments. This explains why the GM algorithm shows slightly better performance with the same number of crashes.

[8]The GM algorithm waits for acknowledgments from a majority of the group; see Section 7.1.2.

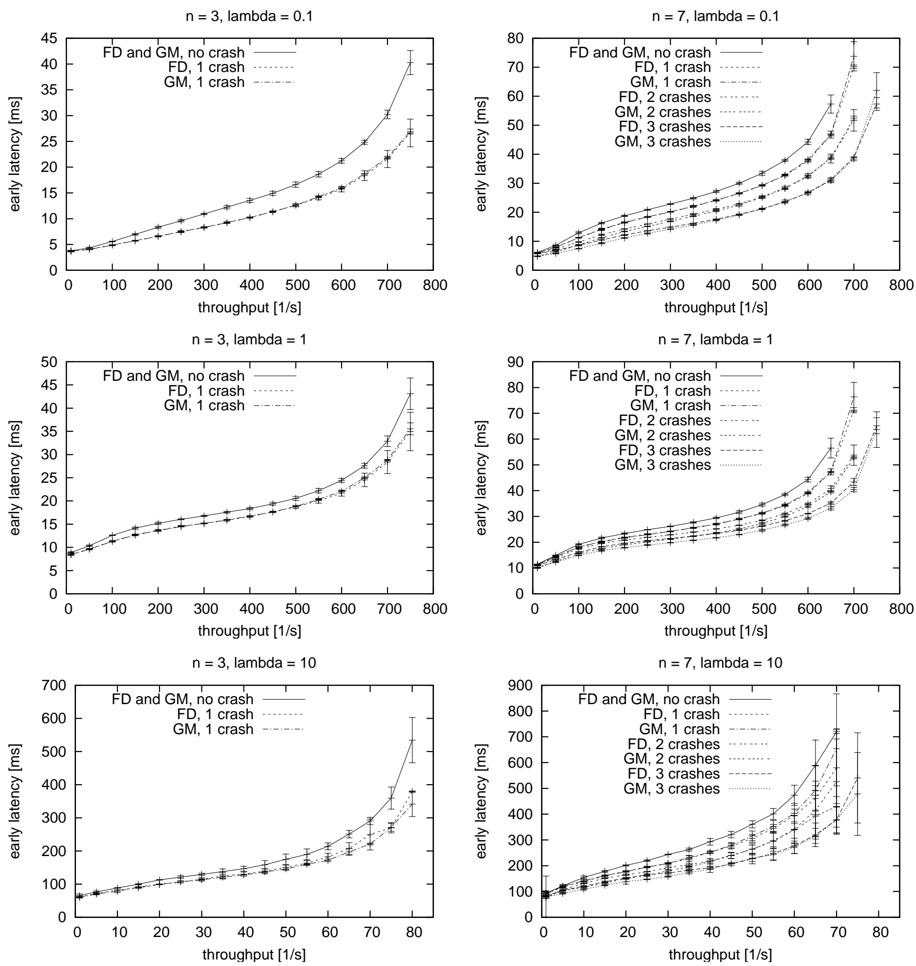


Figure 7.3: Latency vs. throughput with the crash-steady faultload. In each graph, the legend lists the curves from the top to the bottom.

With the GM algorithm, it does not matter which process(es) crash. With the FD algorithm, the crash of the coordinator of Round 1 gives worse performance than the crash of another process. However, the performance penalty when the coordinator crashes is easily avoided: (1) each process tags its consensus proposal with its own identifier, and (2) upon decision, each process re-numbers all processes such that the process with the identifier in the decision becomes the coordinator of Round 1 in subsequent consensus executions. This way, crashed processes will stop being coordinators eventually, hence the steady-state latency is the same regardless of which process(es) we forced to crash. Moreover, the optimization incurs no cost. Hence Fig. 7.3 shows the latency in runs in which non-coordinator processes crash.

Note also that the GM algorithm has higher resiliency on the long term if crashes occur, as the group size decreases with the crashes. E.g., with $n = 7$ and 3 crashes, the GM algorithm can still tolerate one crash after excluding the crashed processes, whereas the FD algorithm can tolerate none.

Suspicion-steady faultload (Figures 7.4 to 7.9). The occurrence of wrong suspicions are quantified with the T_{MR} and T_M QoS metrics of the failure detectors. As this faultload does not involve crashes, we expect that the mistake duration T_M is short. In our first set of results (Fig. 7.4 for $\lambda = 0.1$; Fig. 7.5 for $\lambda = 1$; Fig. 7.6 for $\lambda = 10$) we hence set T_M to 0, and latency is shown as a function of T_{MR}. In each figure, we have four graphs: the left column shows results with 3 processes, the right column those with 7; the top row shows results at a low load (10 s^{-1}; 1 s^{-1} if $\lambda = 10$) and the bottom row at a moderate load (300 s^{-1}; 30 s^{-1} if $\lambda = 10$); recall from Fig. 7.2 that the algorithms can take a throughput of about 700 s^{-1} (70 s^{-1} if $\lambda = 10$) in the absence of suspicions.

The results show that the GM algorithm is very sensitive to wrong suspicions. We illustrate this on Fig. 7.5: even at $n = 3$ and $T = 10$ s^{-1}, that is, the settings that yield the smallest difference, the GM algorithm only works if $T_{MR} \geq 20$ ms, whereas the FD algorithm still works at $T_{MR} = 10$ ms; the latency of the two algorithms is only equal at $T_{MR} \geq 5000$ ms. The difference is greater with all other settings. The reason for the difference is that the GM algorithm has much more work to do than the FD algorithm when a suspicion occurs, as discussed in Section 7.1.4.

In the second set of results (Fig. 7.7 for $\lambda = 0.1$; Fig. 7.8 for $\lambda = 1$; Fig. 7.9 for $\lambda = 10$) T_{MR} is fixed and T_M is on the x axis. We chose T_{MR} such that the latency of the two algorithms is close but not equal at $T_M = 0$. For example, with $\lambda = 1$ (Fig. 7.8), (i) $T_{MR} = 1000$ ms for $n = 3$ and $T = 10$ s^{-1}; (ii) $T_{MR} = 10000$ ms for $n = 7$ and $T = 10$ s^{-1} and for $n = 3$ and $T = 300$ s^{-1}; and (iii) $T_{MR} = 100000$ ms for $n = 7$ and $T = 300$ s^{-1}.

The results show that the GM algorithm is sensitive to the mistake duration T_M as well, not just the mistake recurrence time T_{MR}. The reason is that the greater T_M, the longer it takes to include the wrongly suspected process (p) in the group

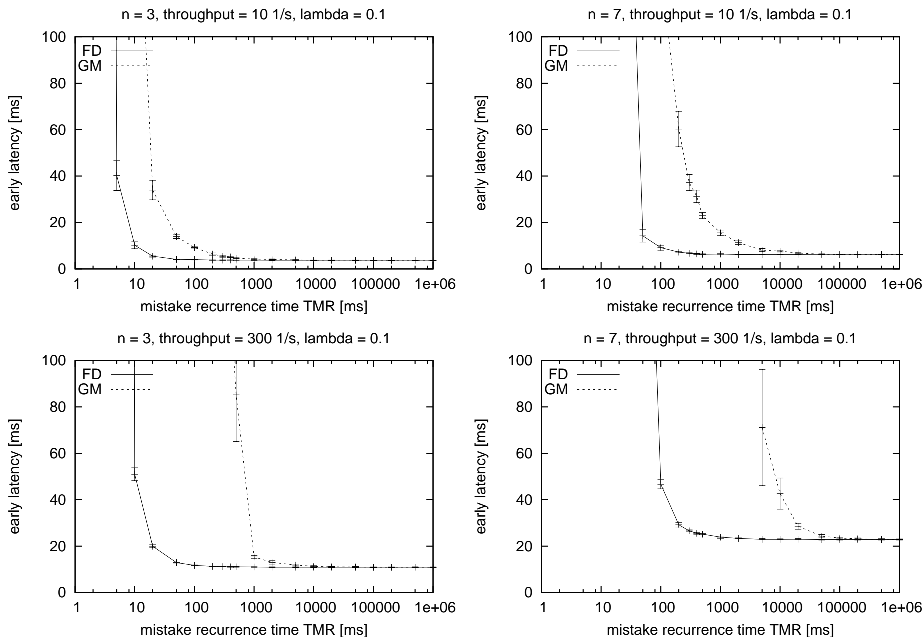


Figure 7.4: Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 0.1$).

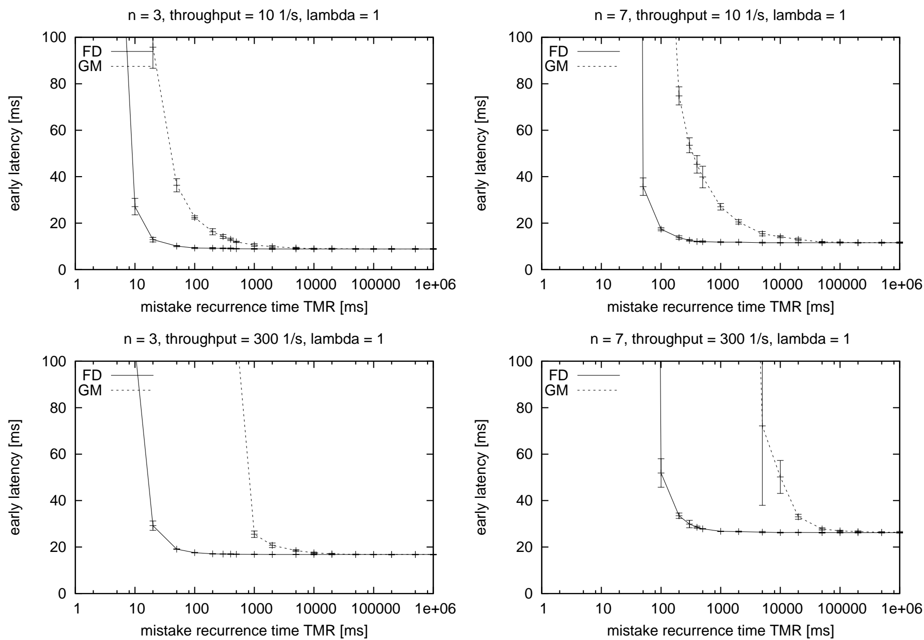


Figure 7.5: Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 1$).

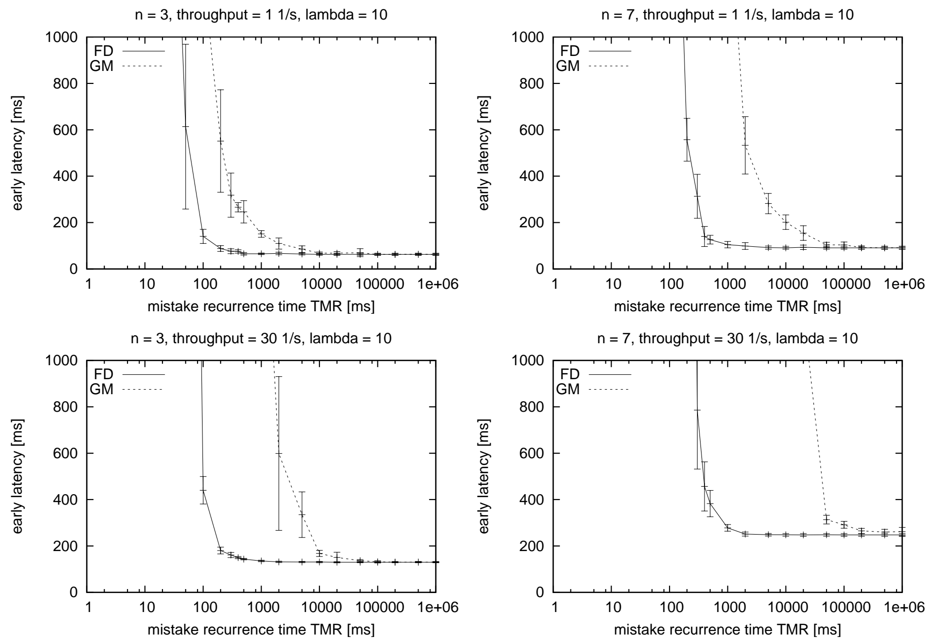


Figure 7.6: Latency vs. T_{MR} with the suspicion-steady faultload, with $T_M = 0$ ($\lambda = 10$).

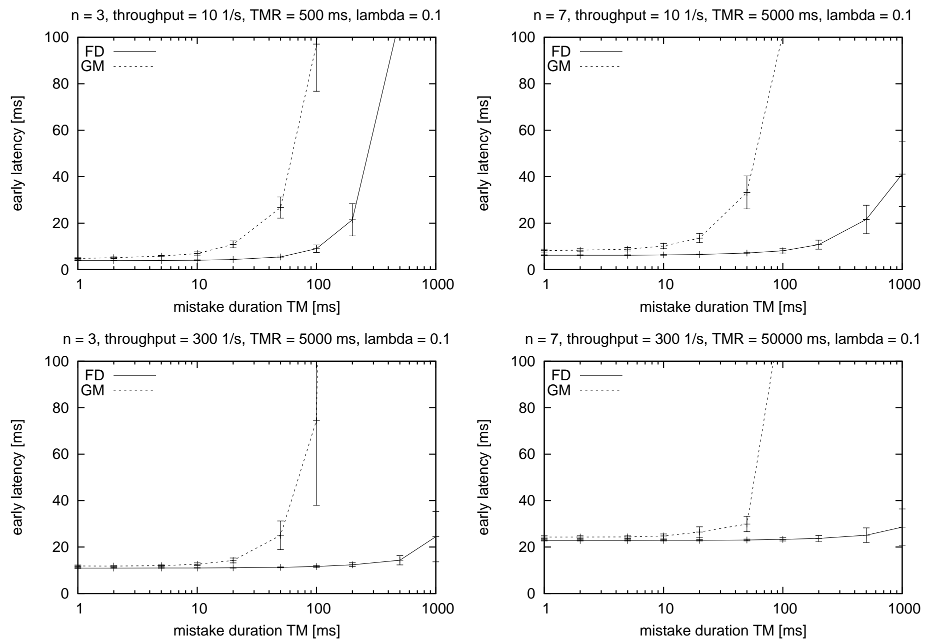


Figure 7.7: Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 0.1$).

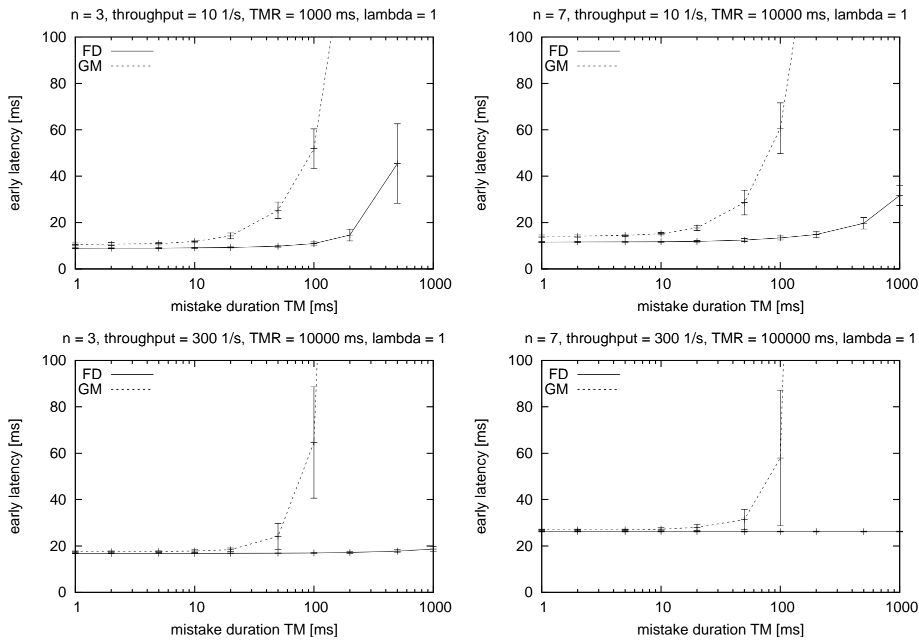


Figure 7.8: Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 1$).

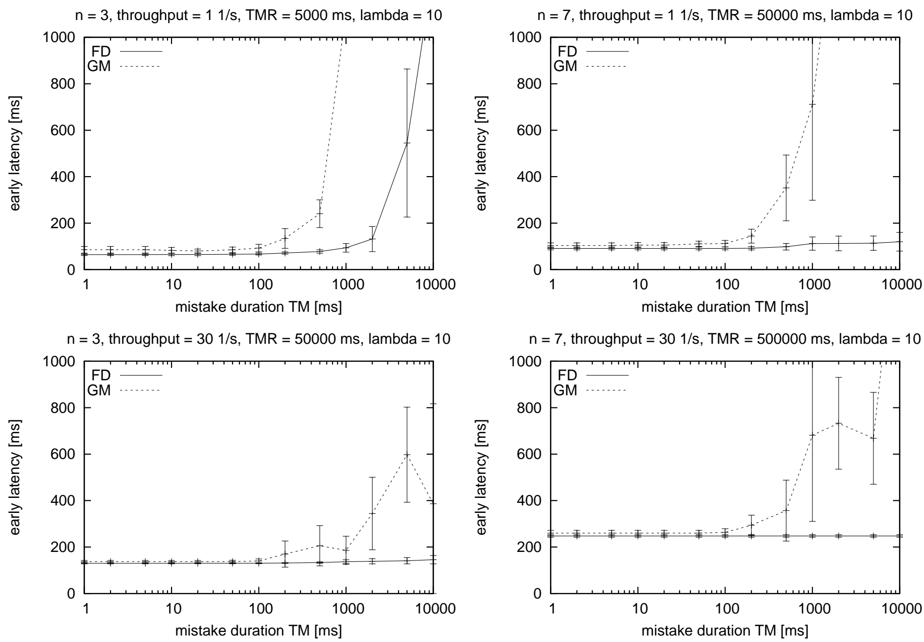


Figure 7.9: Latency vs. T_M with the suspicion-steady faultload, with T_{MR} fixed ($\lambda = 10$).

again. Therefore a lot of broadcasts are generated during the transitional view in which p is excluded. Once p joins the group again, all these messages have to be retransmitted to p; see the description of state transfer in Section 7.1.3. In fact, the number of state transfer messages can be so high that they prevent the second view change from finishing on every process until all state transfer messages are transmitted. This increases the latency of concurrent broadcasts by a great deal. In our implementation, we had to limit the rate of state transfer messages to avoid this increase in latency.

Crash-transient faultload (Fig. 7.10). With this faultload, we only present the latency after the crash of the coordinator and the sequencer, respectively, as this is the case resulting in the highest transient latency (and the most interesting comparison). If another process is crashed, the GM algorithm performs roughly the same, as a view change occurs. In contrast, the FD algorithm outperforms the GM algorithm: it performs slightly better than with the normal-steady faultload (Fig. 7.2), as fewer messages are generated, just like with the crash-steady faultload (Fig. 7.3).

Figure 7.10 shows the *latency overhead*, i.e., the latency minus the detection time T_D, rather than the latency. Graphs showing the latency overhead are more illustrative; note that the latency is always greater than the detection time T_D with this faultload, as no atomic broadcast can finish until the crash of the coordinator/sequencer is detected. The latency overhead of both algorithms is shown for $n = 3$ (left) and $n = 7$ (right) and a variety of values for λ (0.1, 1 and 10 from top to bottom) and T_D (different curves in the same graph).

The results show that (1) both algorithms perform rather well (the latency overhead of both algorithms is only a few times higher than the latency with the normal-steady faultload; see Fig. 7.2) and that (2) the FD algorithm outperforms the GM algorithm with this faultload.

7.4 Discussion

We have investigated two uniform atomic broadcast algorithms designed for the same system model: an asynchronous system (with a minimal extension to allow us to have live solutions to the atomic broadcast problem) and $f < n/2$ process crashes (the highest f that our system model allows). We have seen that in the absence of crashes and suspicions, the two algorithms have the same performance. However, a long time after any crashes, the group membership (GM) based algorithm performs slightly better and has better resilience. With the faultload involving wrong suspicions of correct processes and the one describing the transient behavior after crashes, the failure detector (FD) based algorithm outperformed the GM based algorithm. The difference in performance is much greater when correct processes are wrongly suspected.

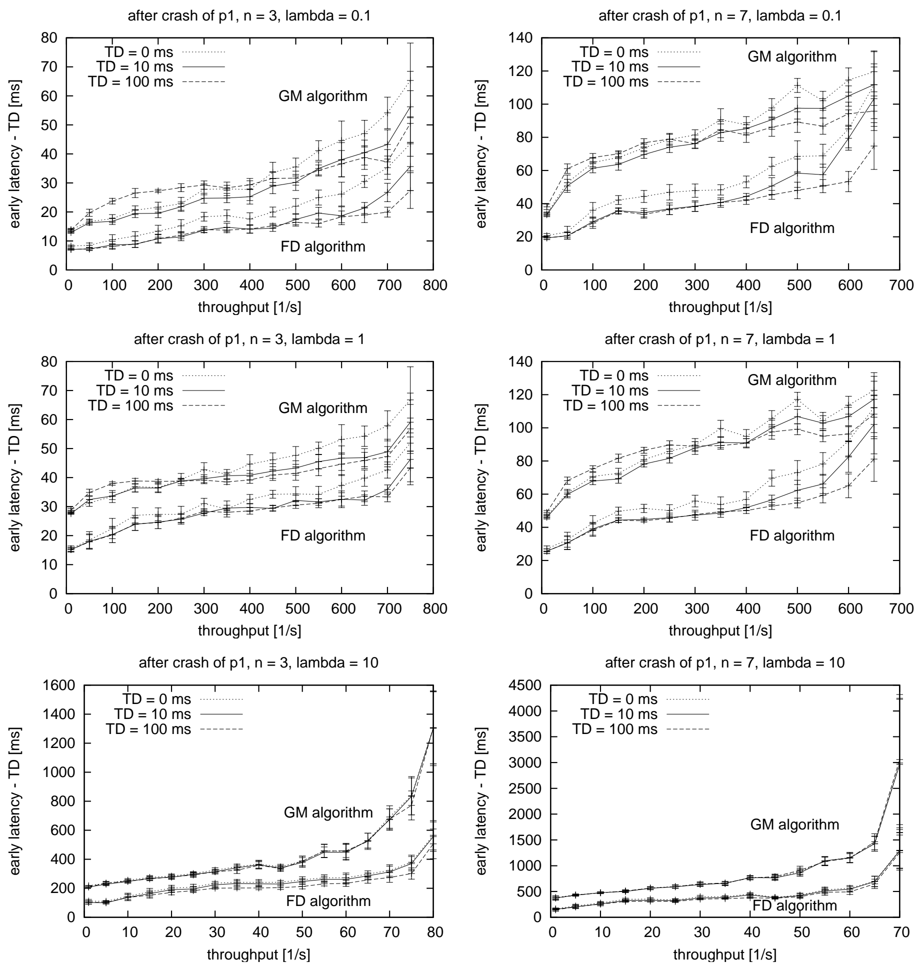


Figure 7.10: Latency overhead vs. throughput with the crash-transient faultload.

Combined use of failure detectors and group membership. Based on our results, we advocate a combined use of the two approaches [CBDS02]. Failure detectors should be used to make failure handling more responsive (in the case of a crash) and more robust (tolerating wrong suspicions). A different failure detector, making fewer mistakes (at the expense of slower crash detection) should be used in the group membership service, to get the long term performance and resiliency benefits after a crash. A combined use is also desirable because the failure detector approach is only concerned with failure handling, whereas a group membership service has a lot of essential features beside failure handling: processes can be taken offline gracefully, new processes can join the group, and crashed processes can recover and join the group. Also, group membership can be used to garbage collect messages in buffers when a crash occurs [CBDS02].

Generality of our results. We have chosen atomic broadcast algorithms with a centralized communication scheme, with one process coordinating the others. The algorithms are practical: in the absence of crashes and suspicions, they are optimized to have small latency under low load, and to work under high load as well (messages needed to establish delivery order are aggregated). In the future, we would like to investigate algorithms with a decentralized communication scheme (e.g., [Lam78, AS00]) as well.

Non-uniform atomic broadcast. Our study focuses on uniform atomic broadcast. What speedup can we gain by dropping the uniformity requirement in either of the approaches (of course, the application must work with the relaxed requirements)? The first observation is that there is no way to transform the FD based algorithm into a more efficient algorithm that is non-uniform: the effort the algorithm must invest to reach agreement on Total Order automatically ensures uniformity ([Gue95] has a relevant proof about consensus). In contrast, the GM based algorithm has an efficient non-uniform variant that uses only two multicast messages (see Fig. 7.1). Hence the GM based approach allows for trading off guarantees related to failures and/or suspicions for performance. Investigating this tradeoff in a quantitative manner is a subject of future work. Also, we would like to point out that, unlike in our study, a state transfer to wrongly excluded processes cannot be avoided when using the non-uniform version of the algorithm, and hence one must include its cost into the model.

Chapter 8

Comparing the performance of two consensus algorithms with centralized and decentralized communication schemes

The two algorithms. In this chapter, we present a comparison study of two consensus algorithms. The one algorithm (due to Chandra and Toueg [CT96]) uses a centralized communication pattern, while the other (due to Mostéfaoui and Raynal [MR99]) uses a decentralized communication pattern. Other aspects of the algorithms are very similar. Both are designed for the asynchronous model with $\Diamond\mathcal{S}$ failure detectors, tolerating $f < n/2$ crash failures. Also, both follow the rotating coordinator paradigm for reaching agreement.

Elements of the performance study. The two consensus algorithms are analyzed in a system in which processes send atomic broadcasts to each other. Since the atomic broadcast algorithm that we use [CT96] leads to the execution of a sequence of consensus to decide the delivery order of messages, evaluating the performance of atomic broadcast is a good way of evaluating the performance of the underlying consensus algorithm in a realistic usage scenario. In our study, the atomic broadcast algorithm uses either of the two consensus algorithms. We study the system using simulation, which allows us to compare the algorithms in a variety of different environments. We model message exchange by taking into account contention on the network and the hosts, using the metrics described in Chapter 5. We use several benchmarks defined in Chapter 6. We evaluate both (1) the steady state latency of atomic broadcast in runs with neither failures nor suspicions and (2) the transient latency after a process crash.

The results. The centralized algorithm requires three communication steps under the most favorable conditions, while the decentralized one needs only two. Hence

it is often believed that the decentralized algorithm is more efficient. Our results show that, contrary to these expectations, the centralized algorithm performs better under a variety of settings. The reason is that the centralized algorithm generates less contention on both the hosts and the network, which often offsets the costs due to the additional communication step.

As the problem of choosing between a decentralized and a centralized variant of an agreement algorithm recurs often in distributed systems (e.g., two and three phase commit protocols have variants of both kinds), we expect that our results are useful in other settings than the ones assumed in this chapter.

Structure. The rest of the chapter is structured as follows. We introduce the algorithms in Section 8.1. Section 8.2 describes all important elements of our performance study. Our results are presented in Section 8.3, and the chapter concludes with a discussion in Section 8.4.

8.1 Algorithms

This section sketches the two consensus algorithms, concentrating on their common points and their differences. We then introduce the atomic broadcast algorithm built on top of consensus.

8.1.1 The consensus algorithms

For solving consensus (defined in Section 2.2.1) we use the Chandra-Toueg $\diamond\mathcal{S}$ algorithm [CT96] and the Mostéfaoui-Raynal $\diamond\mathcal{S}$ algorithm [MR99]. In the sequel, we shall refer to the algorithms as *CT algorithm* and *MR algorithm*, respectively. We also use these names to refer to the atomic broadcast algorithm used with the corresponding consensus algorithm if no confusion arises from doing so.

Common points. The algorithms share a lot of assumptions and characteristics, which makes them ideal candidates for a performance comparison. In particular, both algorithms are designed for the asynchronous model with $\diamond\mathcal{S}$ failure detectors (see Chapter 2 for definitions). Both tolerate $f < n/2$ crash failures. Both are based on the rotating coordinator paradigm: each process executes a sequence of asynchronous rounds (i.e., not all processes necessarily execute the same round at a given time t), and in each round a process takes the role of *coordinator* (p_i is coordinator for rounds $kn + i$). The role of the coordinator is to try to impose a decision value on all processes. If it succeeds, the consensus algorithm terminates. It may fail if some processes *suspect* the coordinator to have crashed (whether the coordinator really crashed or not). In this case, a new round is started.

Execution of a round. In each round of a consensus execution, the CT algorithm uses a centralized communication scheme (see Fig. 8.1) whereas them MR algo-

rithm uses a decentralized communication scheme (see Fig. 8.2). We now sketch the execution of one round in each of the two algorithms. We suppose that the coordinator is not suspected. Further details of the execution are not necessary for understanding the rest of the chapter. Detailed pseudocode descriptions are given in Chapter B, in the Appendix.

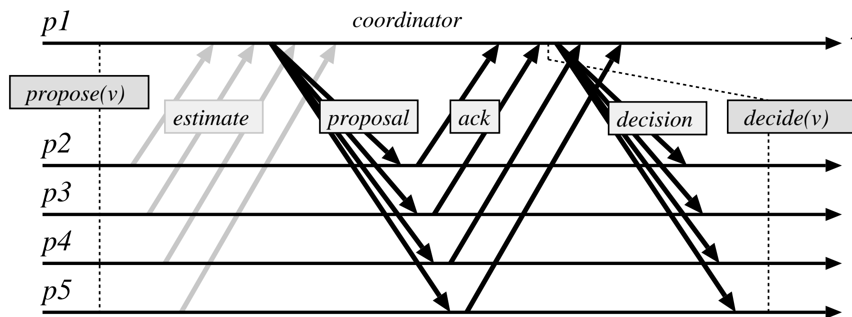


Figure 8.1: Example run of the CT consensus algorithm.

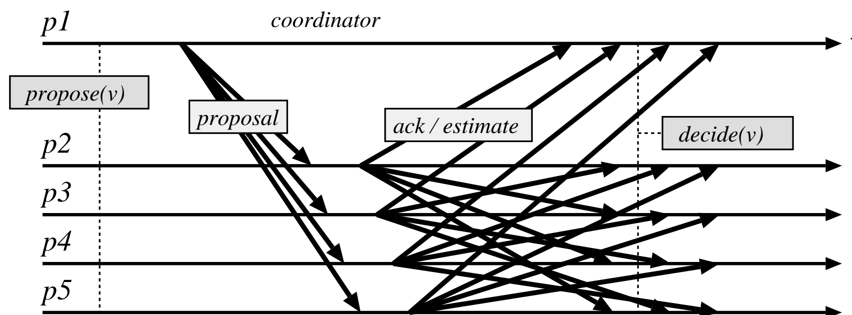


Figure 8.2: Example run of the MR consensus algorithm.

- In the CT algorithm, the coordinator first gathers estimates for the decision value from a majority of processes (*estimate* messages in Fig. 8.1) to choose its proposal from. This phase is only necessary in the second round and later; this is why the messages are grayed out in Fig. 8.1.

- In both algorithms, the coordinator sends a proposal to all (*proposal* messages in Fig. 8.1 and 8.2).

- Upon receiving the proposal, processes send an acknowledgment (*ack* messages). In the CT algorithm, acks are sent to the coordinator only. In the MR algorithm, the ack is sent to all. Moreover, processes in the MR algorithm piggyback their current estimate on the *ack* message, in order to allow the coordinator of the next round to choose a proposal. This is why the MR

algorithm does not require a separate phase to send *estimate* messages. Piggybacking estimates in a similar way is not possible in the CT algorithm, as the coordinator of the next round does not receive the *ack* messages.

- Upon receiving a positive ack from a majority of processes, the coordinator (in the CT algorithm) and all processes (in the MR algorithm) decide. The coordinator in the CT algorithm needs to send its decision to all (decision message in Fig. 8.1). This is not necessary in the MR algorithm, because each process decides independently.

Crashes are handled in the following way: if the coordinator is suspected, the suspecting process sends a negative ack. Negative acks prevent a decision, and lead to another round with a new coordinator.

8.1.2 Optimizations to the consensus algorithms

The consensus algorithms implemented contain several optimizations with respect to the published versions [CT96, MR99]. The goal of the optimizations is to reduce the number of messages if neither crashes nor suspicions occur.

- As mentioned before, the estimate messages are not sent in the first round of the CT algorithm.

- In the original CT algorithm, processes $p_2, \ldots, p_n$ start the second round immediately after sending the ack. This generates unnecessary messages. To prevent this, the coordinator sends an *abort* message if it receives negative acks, and processes wait for the abort message before starting the new round.[1]

- The decision message in the CT algorithm must be sent using reliable broadcast (see Section 2.2.2). We use an efficient algorithm (described in Section B.1 in the Appendix) that requires one broadcast message if the coordinator is not suspected.

- The MR algorithm faces a similar problem: the decision is likely to take place at every process, but this must be verified to ensure the liveness of the algorithm. [MR99] requires that each process sends a decision message upon deciding. These decision messages have a catastrophic effect on performance: in our simulations, the MR algorithm always performed worse than the CT algorithm if these decision messages were used. Our solution is to send decision messages with the efficient reliable broadcast algorithm used to send the decision message in the CT algorithm, but without performing its first step: sending the message to all. In other words, processes that decide

[1] Processes $p_2, \ldots, p_n$ also start a new round if they start suspecting the coordinator.

enter the reliable broadcast algorithm as if they had already received a decision message from the coordinator. This solution ensures that no decision messages are sent unless the coordinator is suspected.

8.1.3 The Chandra-Toueg atomic broadcast algorithm

The Chandra-Toueg atomic broadcast algorithm was introduced in detail in Section 7.1.1 and a pseudocode description of the algorithm is given in Section A.2.7, in the Appendix. We just review its most important features here.

A process executes A-broadcast by sending a message to all processes.[2] When a process receives such a message, it buffers it until the delivery order is decided. The delivery order is decided by a sequence of consensus numbered 1, 2, etc. The initial value and the decision of each consensus is a *set of message identifiers*. Upon receiving a decision, the messages concerned are delivered according to a deterministic function, e.g., according to an order relation defined on their IDs.

The algorithm inherits the system model and any fault tolerance guarantees from the underlying consensus algorithm.

8.2 Elements of our performance study

8.2.1 Performance metrics and workloads

Our main performance metric is the *early latency* of atomic broadcast, i.e., the time that elapses from the broadcast to the first delivery (see Section 6.1). Early latency is simply referred to as latency and is denoted by L. In our study, we compute the mean for L over a lot of messages and several executions.

Latency is always measured under a certain workload. We chose a symmetric workload where (1) the set of senders is all destination processes and (2) the A-broadcast events follow Poisson arrival (see Section 6.2). The workload is characterized by the throughput T, i.e., the overall rate of atomic broadcast messages.

In general, we determine how the latency L depends on the throughput T. Other parameters that influence latency are the choice for the algorithm A and the number of processes n. We also determine the maximal throughput T_{max}.

8.2.2 Faultloads

We evaluate the latency of the atomic broadcast algorithms using two faultloads from Section 6.3. The steady state latency is determined with the normal-steady faultload. This faultload involves neither crashes nor suspicions. The transient latency after a crash is determined with the crash-transient faultload.

In contrast to Chapter 7, the crash-steady and suspicion-steady faultloads are not used. The reason is that, unlike in Chapter 7, even the normal-steady faultload

[2] This message is sent using reliable broadcast. We use an efficient algorithm inspired by [FP01], described in Section B.1, in the Appendix.

offers an interesting comparison. A consequence is that failure detectors only need to be modeled with the detection time parameter T_D (a parameter of the crash-transient faultload).

8.2.3 Modeling the execution environment

Our approach to performance evaluation is simulation, which allowed for more general results as would have been feasible to obtain with measurements in a real system (we can use a parameter in our network model to simulate a variety of different environments). We used the Neko prototyping and simulation framework (see Chapter 4) to conduct our experiments.

We used the model of Section 5.3 for the execution environment. This simple model accounts for resource contention on the network and the hosts. We used both the point-to-point and the broadcast variant of the model for our study. The model has one parameter (λ) that reflects the relative cost of processing on the hosts and the network in the transmission of messages. We used several representative settings for this parameter.

It is worthwhile to describe in detail how crashes appear in our model. If a process p_i crashes at time t, no messages can pass between p_i and CPU_i after t; however, the messages on CPU_i and the attached queues are still sent, even after time t. In real systems, this corresponds to a (software) crash of the application process (operating system process), rather than a (hardware) crash of the host or a kernel panic. We chose to model software crashes because they are more frequent in most systems [Gra86].

8.3 Results

We now present our results for both faultloads and a variety of network models. We obtained results for a variety of representative settings for λ: 0.1, 1 and 10. The settings $\lambda = 0.1$ and 10 correspond to systems where communication generates contention mostly on the network (at $\lambda = 0.1$) and the hosts (at $\lambda = 10$), while 1 is an intermediate setting. For example, in current LANs, the time spent on the CPU is much higher than the time spent on the wire, and thus $\lambda = 10$ is probably the setting closest to reality. We obtained results with both the point-to-point and the broadcast variant of the network model. As these sets of results are quite different, they are presented in separate subsections.

Most graphs show latency vs. throughput (some show latency vs. the number of processes). The maximum throughput T_{max} is approximately the highest throughput value, that is, the x coordinate of the rightmost point, in all graphs showing the steady-state latency; beyond this throughput, the late latency L_{late} (see Section 6.1) did not stabilize. For easier understanding, we set the time unit of the network simulation model to 1 ms. The 95% confidence interval is shown for each point in the graphs.

The two algorithms were always run with an odd number of processes. The reason is that the same number of crash failures k ($k = 1, 2, \ldots$) is tolerated if the algorithms are run with $2k + 1$ and $2k + 2$ processes; thus adding a process to a system with an odd number of processes does not increase the resiliency of the system.

8.3.1 Results in the point-to-point network model

Normal-steady faultload, scalability study (Fig. 8.3). In each graph, latency is shown as a function of the number of processes n. Logarithmic scales are used on both axes, to visualize a big range of latency and to emphasize small values of n. Atomic broadcast are sent at a very low rate (0.1 requests/s). At this throughput, executions of subsequent atomic broadcasts do not influence each other. The parameter λ varies from graph to graph.

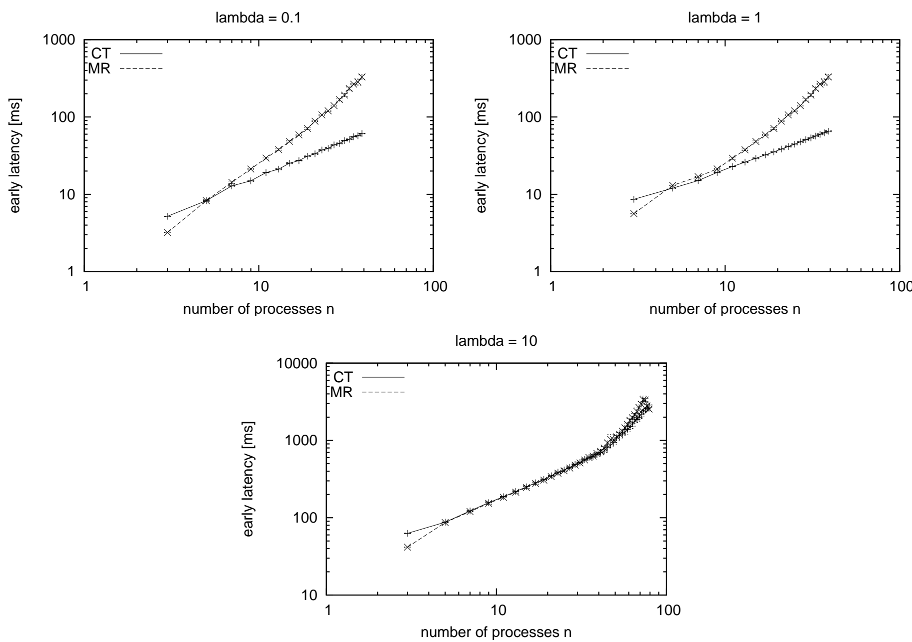


Figure 8.3: Latency vs. number of processes with the normal-steady faultload (point-to-point model).

Each graph can be divided into three regions, depending on the value of n:

- The MR algorithm always performs better at $n = 3$. The reason is that decentralized coordination (MR algorithm) requires one communication step fewer than centralized coordination (CT algorithm; see Figures 8.1 and 8.2). Moreover, this advantage is not offset by the higher resource utilization of the MR algorithm, as it is at higher values of n.

- At very high values of n ($n \geq 9$ if $\lambda = 0.1$; $n \geq 11$ if $\lambda = 1$; approximately $n > 40$ if $\lambda = 10$) the MR algorithm performs much worse. The graphs also show that the latency of the CT algorithm scales linearly with n whereas the latency of the MR algorithm scales quadratically: the slopes of the latency curves in the log-log graphs are about 1 and 2, respectively. The reason is that the CT algorithm uses $O(n)$ messages, whereas the MR algorithm uses $O(n^2)$ messages, though each process only handles $O(n)$ messages in both algorithms. This makes the MR algorithm network bound at high values of n, and the effect of a quadratic number of messages shows directly.

- At intermediate settings for n, the two algorithms perform roughly the same. The reason is that the higher resource utilization of the network resource starts to show (unlike at $n = 3$) but both algorithms are still CPU bound (unlike at high values of n).

Normal-steady faultload, algorithms under load (Figures 8.4 to 8.6). Nine latency vs. throughput graphs are shown, three in each figure. The parameter λ changes from figure to figure, and each graph within a figure shows results for $n = 3, 5$ and 7, respectively.

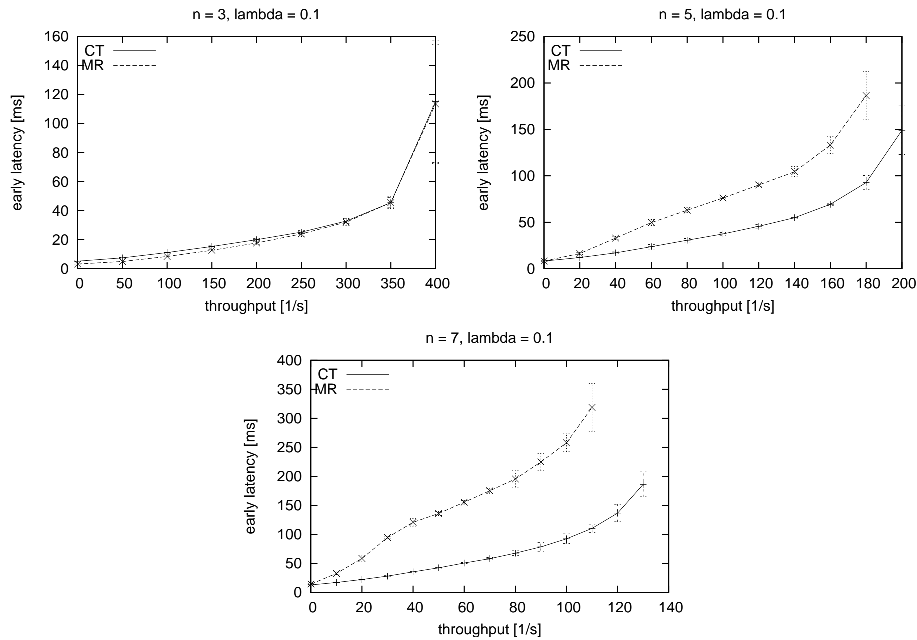


Figure 8.4: Latency vs. throughput with the normal-steady faultload ($\lambda = 0.1$, point-to-point model).

One can observe two different behaviors:

- The CT algorithm has worse performance at $n = 3$, and also at $n = 5$

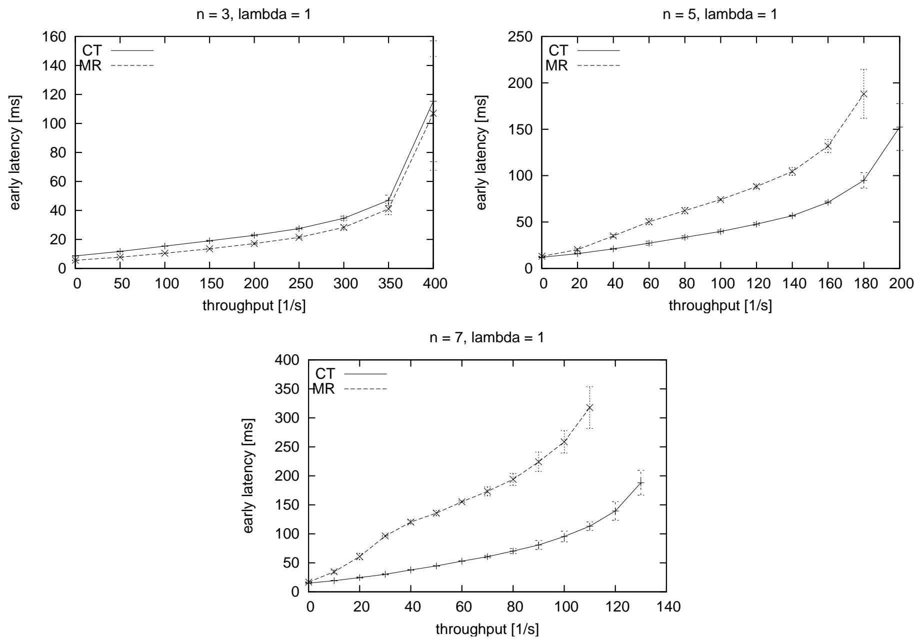


Figure 8.5: Latency vs. throughput with the normal-steady faultload ($\lambda = 1$, point-to-point model).

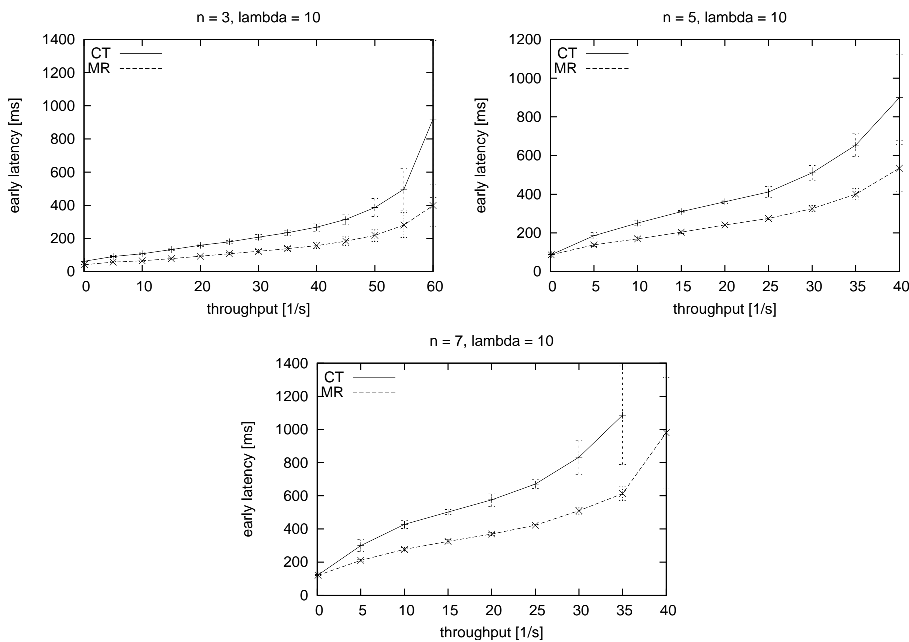


Figure 8.6: Latency vs. throughput with the normal-steady faultload ($\lambda = 10$, point-to-point model).

and 7 when $\lambda = 10$. The relative performance difference grows with λ at $n = 3$. The performance difference is roughly proportional to the throughput at $\lambda = 10$; at $\lambda = 0.1$ and 1, the difference is small.

The reason for this behavior is that the CT algorithm loads the coordinator much more than the MR algorithm: beside providing a proposal, it also must collect acks and send the decision, as shown in Fig. 8.1 (the coordinator has the most loaded CPU with both algorithms). This explanation holds when the network utilization is the same in both algorithms ($n = 3$: the same number of unicast messages pass on the network) or the cost of processing a message on the CPUs is much higher than processing it on the network ($\lambda = 10$).

- The MR algorithm has worse performance at $n = 5$ and 7 when $\lambda = 0.1$ or 1. The performance difference is roughly proportional to the throughput. The relative performance difference grows with n.

  The reason for this behavior is that the load on the CPUs does not matter with these settings for n and λ, unlike in the previous case. Instead, the determining factor is that the MR algorithm loads the network more; the higher n, the more the network is loaded. Also, increasing the throughput leads to higher queuing times in the network buffers of the model (see Section 5.3).

Crash-transient faultload (Figures 8.7 to 8.9). With this faultload, we only present the latency after the crash of the coordinator, as this is the case resulting in the highest transient latency (and the most interesting comparison). If another process is crashed, both algorithms offer roughly the latency observed with the normal-steady faultload.

The figures show the *latency overhead*, i.e., the latency minus the detection time T_D, rather than the latency. Graphs showing the latency overhead are more illustrative; note that the latency is always greater than the detection time T_D with this faultload, as no atomic broadcast can finish until the crash of the coordinator is detected. The arrangement of the figures and graphs is the same as with the normal-steady faultload, with the throughput on the x axis.

We set the failure detection timeout T_D to 100 ms at $\lambda = 0.1$ or 1 and to 1000 ms at $\lambda = 10$. This choice models a reasonable trade-off for the failure detector: the latency overhead is comparable to T_D, to make sure that the failure detector does not degrade performance catastrophically when a crash occurs. On the other hand, the detection time is high enough (in most networking environments) to avoid that failure detectors suspect correct processes.

The results are very similar to the previous set of results, as can be seen by comparing Fig. 8.4 with Fig. 8.7; Fig. 8.5 with Fig. 8.8; and Fig. 8.6 with Fig. 8.9. The same observations and explanations apply.

8.3.2 Results in the broadcast network model

Normal-steady faultload, scalability study (Fig. 8.10). In each graph, latency is shown as a function of the number of processes n. Linear scales are used on both axes. Atomic broadcast are sent at a very low rate (0.1 requests/s). The parameter λ varies from graph to graph.

One can see that the MR algorithm offers a slightly lower latency. Moreover, the difference in latency does not depend on n. The reason is that in the broadcast model, the MR algorithm terminates in one communication step fewer, and that the most heavily loaded resources (the network and the CPU of the coordinator) process one message fewer per consensus (n with the MR algorithm and $n + 1$ with the CT algorithm).

Normal-steady faultload, algorithms under load (Figures 8.11 to 8.13). Just as before, nine latency vs. throughput graphs are shown, three in each figure. The parameter λ changes from figure to figure, and each graph within a figure shows results for $n = 3, 5$ and 7, respectively.

The MR algorithm performs better at any load. The reason is that in the broadcast model, the most heavily loaded resources (the network and the CPU of the coordinator) process one message fewer per consensus (n with the MR algorithm and $n + 1$ with the CT algorithm).

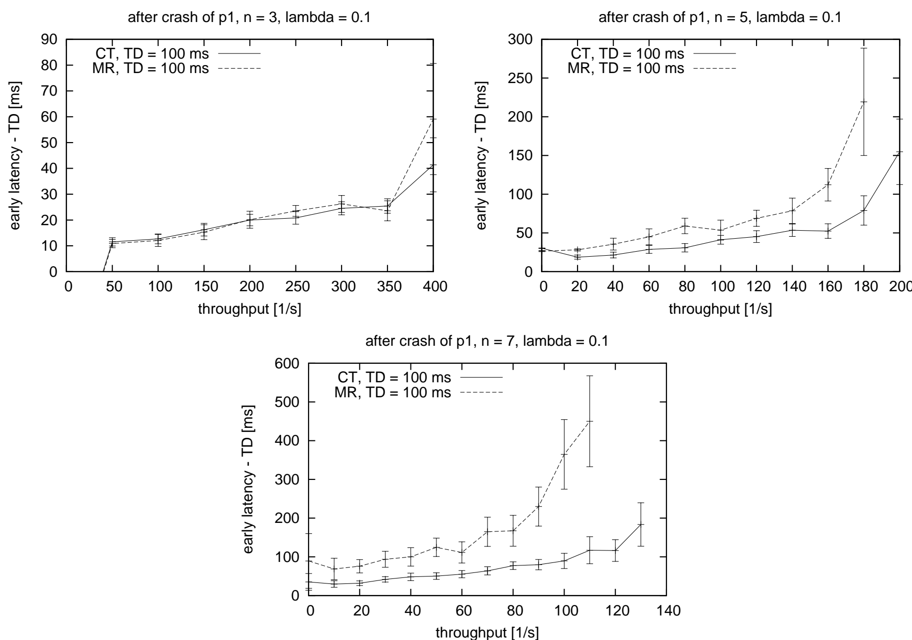


Figure 8.7: Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 0.1$, point-to-point model).

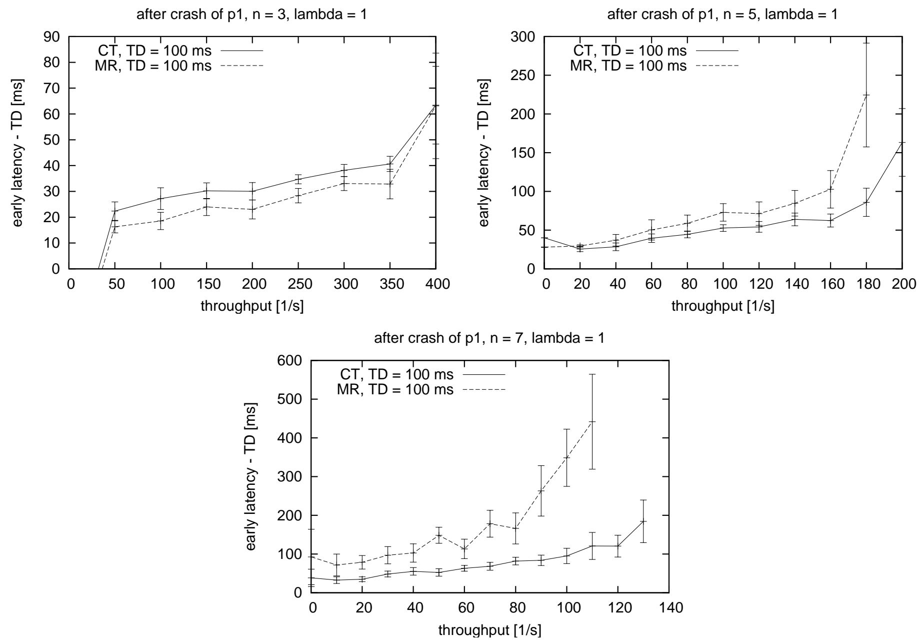


Figure 8.8: Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 1$, point-to-point model).

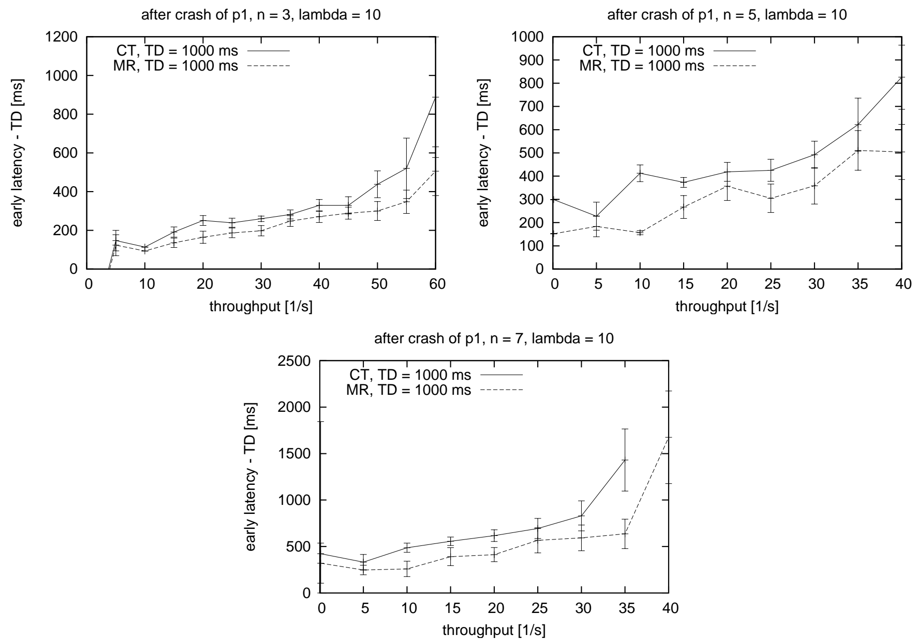


Figure 8.9: Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 10$, point-to-point model).

Crash-transient faultload (Figures 8.14 to 8.16). Once again, we only present the latency after the crash of the coordinator, as this is the case resulting in the highest transient latency (and the most interesting comparison). The figures show the *latency overhead*. The arrangement of the figures and graphs is the same as with the normal-steady faultload with the throughput on the x axis. Just as with the point-to-point model, we set the failure detection timeout T_D to 100 ms at $\lambda = 0.1$ and 1 and to 1000 ms at $\lambda = 10$.

With this faultload, the performance of the CT algorithm is much worse, at all settings of n and λ. The reason is that, in addition to the differences observed with the normal-steady faultload, the CT algorithm takes one communication step more (the first phase of the second round; see the gray *estimate* messages in Fig. 8.1) than the MR algorithm. These *estimate* messages are piggybacked on *ack* messages in the MR algorithm, as discussed in Section 8.1.1. The fact that piggybacking is possible is an advantage of the decentralized structure of the MR algorithm.

8.4 Discussion

We have investigated two consensus algorithms designed for the same system model: an asynchronous system (with a minimal extension to allow us to solve the consensus problem) and $f < n/2$ process crashes (the highest f that our system model allows). Also, both algorithms are based on the rotating coordinator

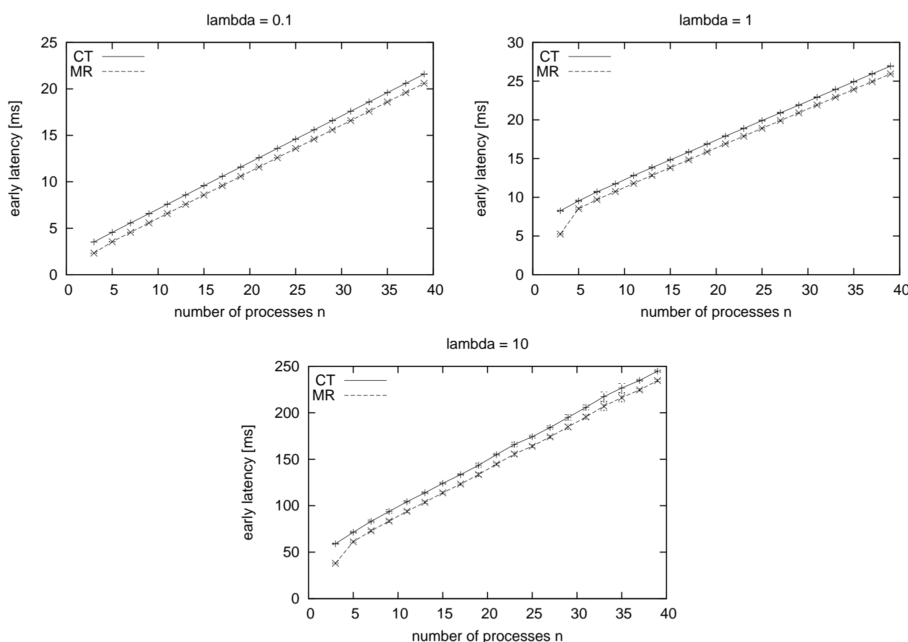


Figure 8.10: Latency vs. number of processes with the normal-steady faultload (broadcast model).

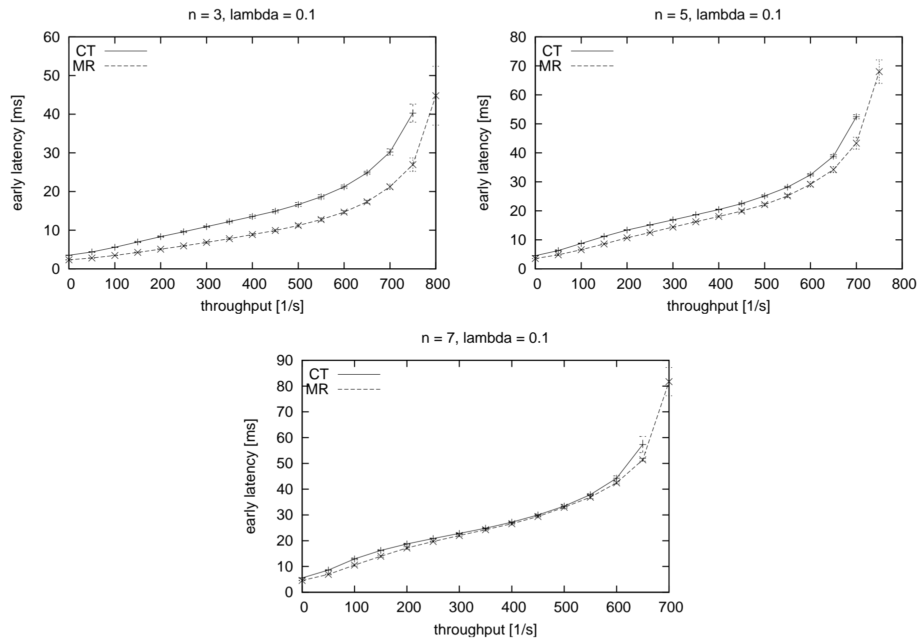


Figure 8.11: Latency vs. throughput with the normal-steady faultload ($\lambda = 0.1$, broadcast model).

paradigm. The main difference is that, in each round, the CT algorithm uses a centralized and the MR algorithm a decentralized communication pattern.

Summary of the results. We now summarize the results of the study as a list of observations. These observations can be used by implementors when deciding which algorithm to deploy in a given system.

1. We found that in a network model with point-to-point messages only, the MR algorithm performs much worse both when the number of processes n or the load on the system is high. The reason is that the MR algorithm generates much more contention on the network.

2. In a network model with broadcast messages, the MR algorithm performs slightly better. The difference in latency does not depend on the number of processes.

3. In a network model with broadcast messages, the MR algorithm reacts much faster to failures. The reason is that its decentralized nature allows it to piggyback information in order to save a communication step when a crash failure occurs.

4. Frequently, only one crash failure needs to be tolerated. If this is the case, i.e., the consensus algorithm runs on three processes, the MR algorithm is

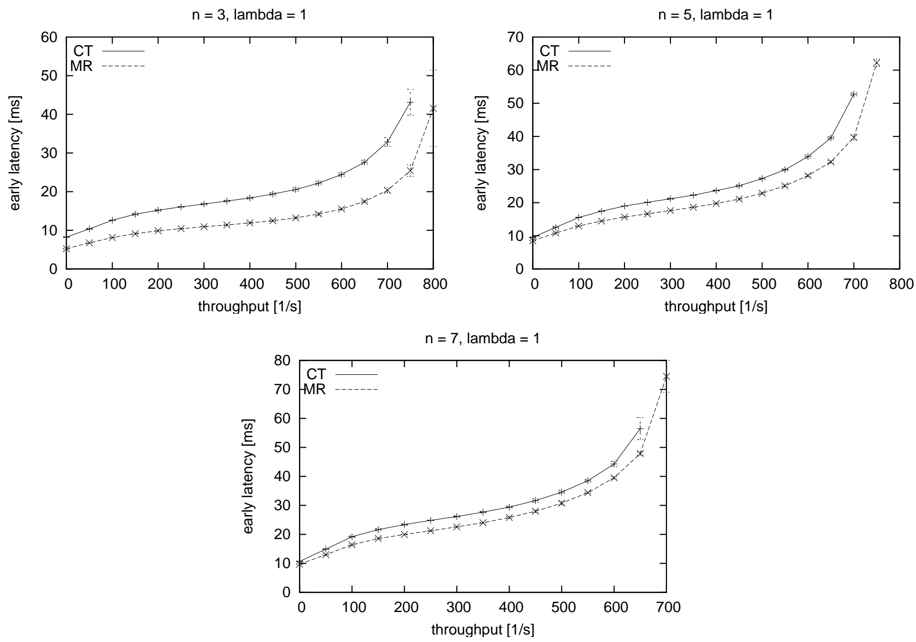


Figure 8.12: Latency vs. throughput with the normal-steady faultload ($\lambda = 1$, broadcast model).

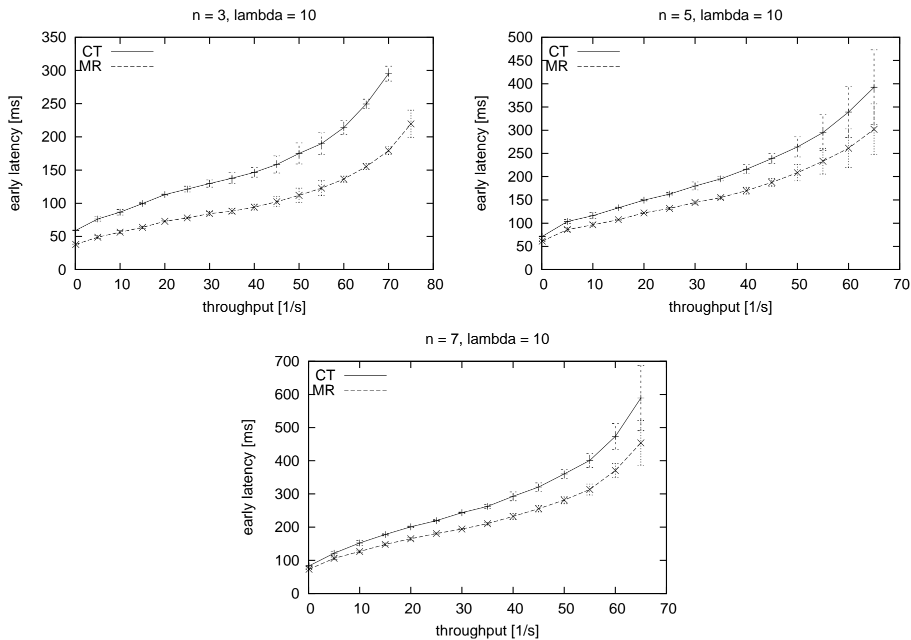


Figure 8.13: Latency vs. throughput with the normal-steady faultload ($\lambda = 10$, broadcast model).

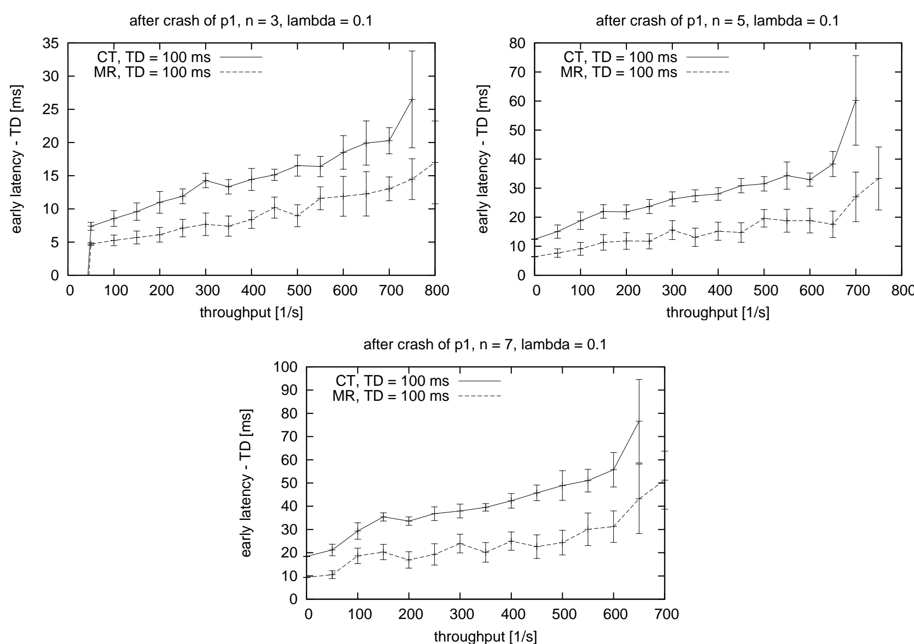


Figure 8.14: Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 0.1$, broadcast model).

a better choice regardless of whether the network supports broadcast messages.

On the use of time complexity. Time complexity predicts that the MR algorithm performs much better: it only needs 2 communication steps to decide, whereas the CT algorithm needs 3. Even our simple model shows that this does not always hold: Observations 1 and 2 cannot be explained using time complexity (in the case of Observation 2, the MR algorithm indeed performs better, but the performance difference is really small). The phenomenon that time complexity misses and our model includes is the contention on system resources generated by message passing.

Early decision schemes. In the MR algorithm as described in [MR99], a process sends a decision message to all when it decides. The authors call this an early decision scheme; the underlying intuition is that this scheme shortcuts the execution of the algorithm, as a process can decide when it sees a decision message. We found that, contrary to its name, the early decision scheme delays decisions, by loading the hosts and the network with messages (see Section 8.1.1). This is not surprising: the early decision scheme generates $O(n^2)$ messages in a network with point-to-point.messages only. The lesson is that intuition is often misleading when it is based on a view of the system that does not take resource contention into account.

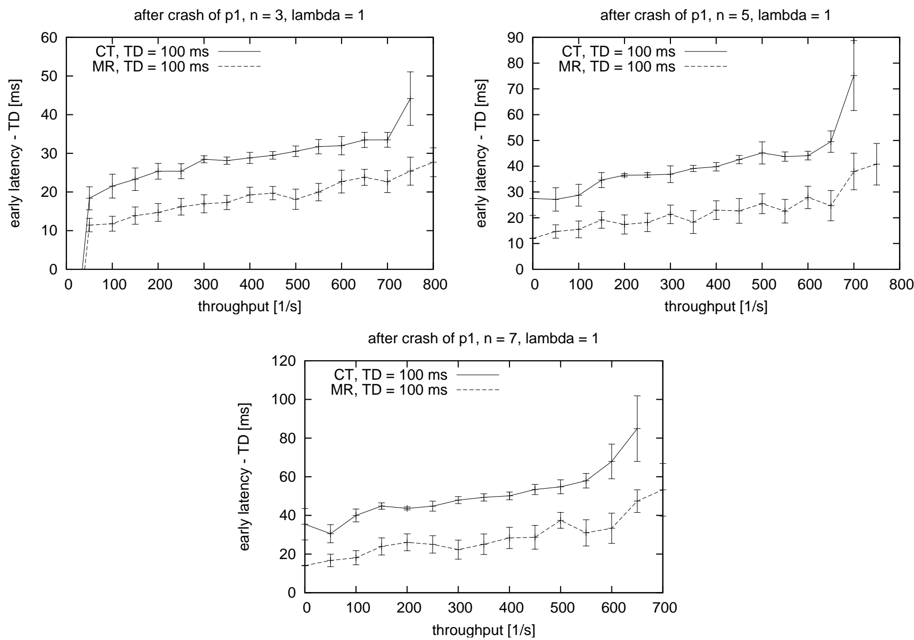


Figure 8.15: Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 1$, broadcast model).

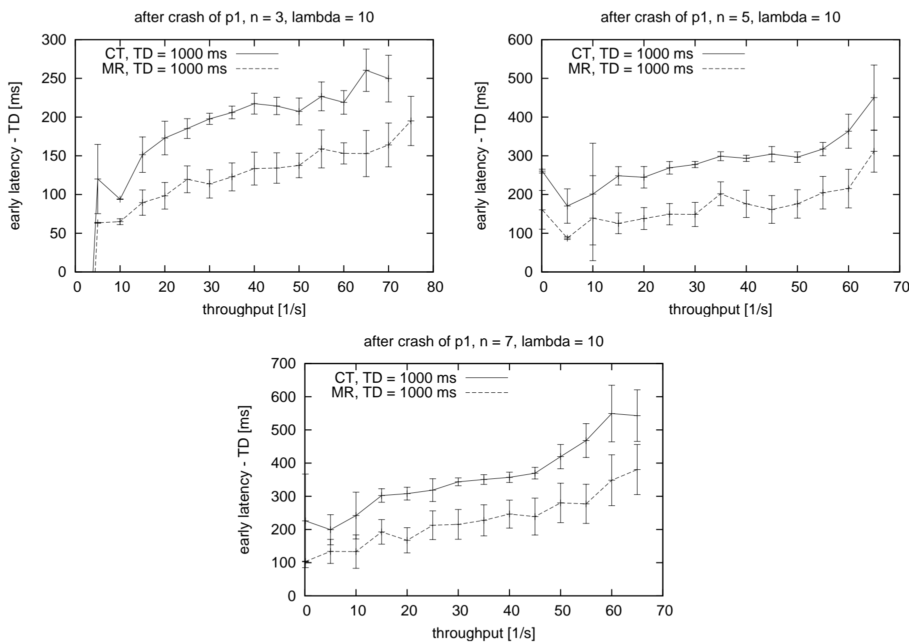


Figure 8.16: Latency overhead vs. throughput with the crash-transient faultload ($\lambda = 10$, broadcast model).

Chapter 9

Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be?

Fault tolerance can be achieved in distributed systems by replication. However, Fischer, Lynch and Paterson have proven an impossibility result about consensus in the asynchronous system model [FLP85]. Similar impossibility results have been established for atomic broadcast and group membership, and should be as such relevant for implementations of a replicated service. However, the practical impact of these impossibility results is unclear. For instance, do they set limits to the robustness of a replicated server exposed to extremely high loads?

This chapter tries to answer this question by describing an experiment conducted in a local area network (LAN). It consists of client processes that send requests to a replicated server (three replicas) using an atomic broadcast primitive. The experiment has parameters that allow us to control the load on the system and the failure detection time offered by the failure detectors underlying the replicated server.

Our main observation is that the replicated server never stops processing requests, not even at arbitrarily high load and very small failure detection times (1 ms). The result was surprising to us, as we expected that our atomic broadcast algorithm would stop delivering messages at such small failure detection times. So, by trying to illustrate the practical impact of impossibility results, we discovered that we had implemented a very robust replicated service.

The rest of the chapter is structured as follows. Section 9.1 presents the motivation for our work and summarizes the experiment and the results. Section 9.2 presents related work. Section 9.3 introduces the algorithms used in the experiment. Section 9.4 describes the environment and some features of the implementation. Section 9.5 explains how we tested the robustness of the replicated server. Section 9.6 describes in detail the results obtained in our experiments. Finally, Section 9.7 discusses these results.

9.1 Introduction

High availability is often achieved by the replication of components or services. Although replication is an intuitive and readily understood concept, its implementation is difficult. Replicating a service in a distributed system requires that the states of all replicas of the service are kept consistent, which can be ensured by a specific replication protocol [Sch90, BMST93]. A replication protocol is typically implemented using group communication primitives, e.g. atomic broadcast (see Section 2.2.3).

However, Fischer, Lynch and Paterson have proven an impossibility result for consensus in the asynchronous system model [FLP85], a result commonly known as the FLP impossibility result (see Section 2.2.5).[1] The impossibility result also applies to atomic broadcast [CT96]. The impossibility of group membership — another problem related to replication — in asynchronous systems was also established [CHTCB96]. Formally, these impossibility results set a limit on the level of robustness that a replicated service can achieve. Practitioners, however, disregard these impossibility results, i.e., they consider them of no practical relevance. The reason is that real systems usually exhibit some level of synchrony, i.e., they are not exactly asynchronous. Consequently, the implications of the impossibility result to real systems are difficult to see, and these theoretical results are largely ignored in practice.

On the other hand, no paper in the literature refers to experiments in which the implementation of replication is exposed to extremely high loads. How robust can a system be under these conditions? Do high loads actually prevent the system from making progress (as stated by the FLP impossibility result), and so limit the robustness of the system? How robust can a fault tolerant server be?

To answer these questions, we designed an experiment for a local area network (LAN). It consists of client processes that send requests to a replicated server using an atomic broadcast primitive. The experiment has a parameter which specifies the load on the system (the rate of requests coming from the clients). The other parameter is the timeout used by our heartbeat failure detectors. The implementation is such that this timeout is an upper bound on the failure detection time offered by the failure detectors: the smaller the timeout, the faster the failure detection. It is clear that if we use very slow failure detection, say with a detection time of one minute, our system could be extremely robust, as false failure suspicions would be avoided with a high probability. However, the behavior of our system in the case of a crash would be disastrous: with a detection time of one minute, the response time could be extremely high (if the crash affects a process which has an important role at the moment of the crash). In order to avoid such robust, but badly performing systems, we were decreasing the detection time values (and increasing the frequency of sending heartbeats). A side effect is that the failure detectors make more and more

[1] An asynchronous system — which models a system with unpredictable CPU and channel loads — is a system in which there is no assumption neither on message communication delays nor on relative speeds of processes. See also Section 2.1.1.

mistakes by suspecting correct processes. Our intuition was that, as we decrease the detection time (and increase the frequency of heartbeats), the atomic broadcast algorithm would stop making progress at some point in its execution. Interestingly, our experiment showed that this was not the case: up to very small detection times (i.e., 1 ms) and at arbitrarily high load conditions, the atomic broadcast algorithm never stops delivering messages (i.e., it always works). Thus, by challenging our implementation with high loads and small failure detection times, we discovered that we had implemented a replicated service which is extremely robust in a LAN.

9.2 Related work

To the best of our knowledge, no paper in the literature refers to experiments in which the implementation of replication is exposed to extremely high loads. Nevertheless, there are two papers [CF99, CTA02] which implicitly suggest ways of implementing extremely robust replicated servers, and support their arguments by performance evaluation results.

[CF99] introduces the timed asynchronous system model for distributed algorithms. This model, extended with what the authors call progress assumptions, allows them to solve consensus. Therefore, if the timed asynchronous model matches reality, one can build extremely robust replicated servers using algorithms developed for this model. The authors support the assumptions of their model by an extensive set of measurements performed in a LAN. They validate the assumptions of their core model even under high load. However, the progress assumptions (which make the model powerful enough to solve consensus) are validated only under moderate load (1/4 of the network capacity).

[CTA02] presents a failure detector based on heartbeats and proves that this algorithm is optimal (in terms of the quality of service measures defined in the paper and in the class of heartbeat failure detectors), for any kind of distribution for the message delays. The paper also gives an adaptive version of the algorithm that approximates the optimum even if the distribution of message delays is not known in advance or changes over time. The results are supported by analytical computations and a simulation study. Indeed, such a failure detector would be ideally suited to implement a replicated server which is extremely robust in a variety of environments and for a variety of loads. However, the paper assumes that message delays are *independent* random variables. This assumption is far from being true for two subsequent messages if the network is under high load (as we saw in our experiment when logging messages): if a message suffers a high delay, usually the next message suffers a comparably high delay as well.

9.3 Algorithms

Active replication. Our experiment consists of a replicated server and several clients. Each client repeatedly sends a request to the replicated server and waits for

a reply. The server is replicated by means of *active replication* (also called *state machine approach*) [Sch90, Sch93, Pol94]. In active replication, clients use atomic broadcast to send their requests to the replicas. Atomic broadcast ensures that all server replicas receive the client requests in the same order (see Section 2.2.3). Upon reception of a request, each server replica performs the same deterministic processing (in our case, writing a number to a file) and sends back a reply to the client. The client waits for the first reply, and ignores all further replies to the same request.

Atomic broadcast. The atomic broadcast algorithm used is due to Chandra and Toueg [CT96]. This algorithm was introduced in detail in Section 7.1.1 and a pseudocode description of the algorithm is given in Section A.2.7, in the Appendix. We just review its most important features here.

The algorithm solves atomic broadcast by executing a sequence of consensus, where each consensus decides on a set of messages to be delivered. The algorithm inherits the system model and any fault tolerance guarantees from the underlying consensus algorithm.

Consensus. We use the Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm [CT96] (see Section 2.2.1 for a definition of the consensus problem), an algorithm designed for the asynchronous system model with the failure detector $\diamond\mathcal{S}$ and a majority of correct processes. We only give an overview of the algorithm here, explaining the parts necessary for understanding the rest of the chapter. A detailed pseudocode description is given in Section B.2, in the Appendix.

The algorithm is based on the rotating coordinator paradigm. Processes proceed in consecutive asynchronous rounds (*not* all processes are necessarily in the same round at a given time). In each round a predetermined process acts as the coordinator. The coordinator proposes a value for the decision. A round succeeds if a decision is taken in that round; if some process decides (and does not crash) it forces the other processes to decide, and thus the algorithm is guaranteed to terminate shortly. A round might fail when its coordinator crashes, or when its coordinator, while correct, is suspected by other processes. Consensus might terminate in a single round, i.e., the first round can already succeed. Some runs might require more rounds, though; in general, the more often the coordinator is suspected, the more rounds the algorithm will take to terminate.

Failure detection. The consensus algorithm relies on a failure detection mechanism implemented using heartbeat messages (Fig. 9.1). Each process periodically sends a heartbeat message to all other processes. Heartbeat messages are timestamped with the time of sending. Failure detection is parameterized with a timeout value T and a heartbeat period T_h. When a process p receives a heartbeat message from process q with timestamp t, it trusts q up to time $t+T$, and then starts suspecting q to have crashed (unless a more recent heartbeat message arrives). Application

messages are timestamped also with the time of sending, and are used as heartbeat messages.

Failure detectors can be characterized by the quality of service metrics introduced in [CTA02], and extensively discussed in Section 6.3.4. One of these metrics is the *detection time* T_D, characterizing the speed of detecting failures: T_D is the time that elapses from p's crash to the time when q starts suspecting p permanently (see Fig. 6.2). Our failure detector implementation is such that it ensures a detection time of $T_D = T$ or better: if a process p crashes at time t_c, all failure detectors will permanently suspect p at time $t_c + T$. The reason is simply that all heartbeats coming from p are timestamped with a time earlier than t_c.

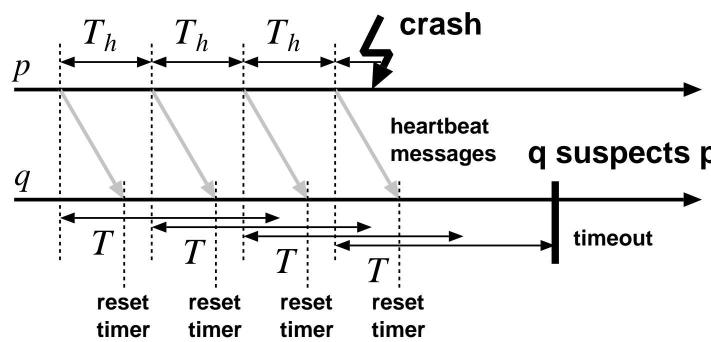


Figure 9.1: Heartbeat failure detection.

9.4 Environment and implementation issues

9.4.1 Environment

The experiment described in the previous section was run on a cluster of 13 PCs running Red Hat Linux 7.2 (kernel 2.4.18). The hosts have Pentium III 766 MHz processors and 128 MB of RAM, and are interconnected by a 100 Base-TX duplex Ethernet hub. Three server replicas were used, as three is the minimum number of replicas for which the atomic broadcast algorithm used tolerates process crashes. Each server replica ran on a different host, while the remaining 10 hosts were used for the clients (there was more than one client per host). We had to use a lot of hosts, otherwise the client hosts turned out to be a bottleneck and thus we could not generate a sufficiently high rate of client requests. The algorithms were implemented in Java (Sun's JDK 1.4.1_01) on top of the Neko framework (see Chapter 4).

9.4.2 Communication protocols

We have three types of messages with different delivery requirements in our system: (1) heartbeat messages, (2) client requests, as well as (3) messages between

server replicas and replies to clients. We use the UDP protocol for transmitting heartbeat messages, for the loss of a heartbeat message is not critical. The other messages need reliable transmission, therefore the straightforward choice is the TCP protocol (nevertheless, we chose UDP rather than TCP for client requests, for reasons discussed in the next paragraph). However, TCP has problems with extreme overload situations. In such situations, two hosts can be partitioned from each other for a long time, and TCP connections break, for a lot of retransmissions fail in a row. The number of times TCP tries to retransmit a packet is given by the parameter `tcp_retries2` of the Linux TCP implementation. We solved the problem by setting the parameter from the default value (15) to a very high value, for all the hosts involved.[2]

We could not use TCP for transporting client requests. To understand why, recall that the goal of our experiment was to investigate the behavior of our system under arbitrarily high loads. Therefore we had to avoid that flow control decreases the load on the system, i.e., the rate of client requests.[3] In the context of our experiment (many clients on one host), the implication is that we cannot use a single TCP connection per host for the client requests: in this case, TCP's congestion control mechanism makes sure that the network never gets overloaded. We cannot use one TCP stream per client, either, for the servers would have to handle a huge number of simultaneous connections, more than the operating system allows. We have the same problem if we use one TCP connection for each request (and send the reply on another connection). The remaining choice for transporting client requests is UDP.[4] This way, the network can be arbitrarily loaded with client requests.

9.4.3 Flow control in the application

Flow control is an essential mechanism in distributed systems: it ensures that components do not receive more work than they can handle. Any non-trivial system needs flow control, but a lot of systems can rely on flow control offered by TCP. This was not sufficient for our system. We explain the reasons below and then present our flow control mechanism (implemented within the application layer).

Systems that rely on TCP use a send primitive that blocks whenever TCP's sending buffer fills up (e.g., because the receiver is slow). A blocking send primitive only works well for client-server interactions; they constitute a poor way of synchronizing more complex distributed systems. In our case, blocking sends led to deadlocks: under high load, all server replicas got blocked in their send operations and could not receive messages to resolve the deadlock. Using threads dedicated to receiving messages does not solve the problem. No deadlocks appear,

[2]Another possibility is to use a session layer protocol which re-establishes the connection and ensures that messages are delivered exactly once. Such a protocol is described in [EUS03].

[3]Otherwise, we would have constructed a controlled environment which includes the replicated service, *as well as its clients*. The high load scenarios we are interested in would not occur at all in such an environment.

[4]Retransmitting lost or dropped client requests is the responsibility of the client.

but — depending on the exact implementation — the number of threads or the size of message queues continues growing until system resources are exhausted.

We solved the problem by adding an *outgoing message queue* in the application layer. Send operations which are non-blocking deposit messages in that queue, and a dedicated thread empties the queue and performs the TCP send operation. Without flow control, the outgoing message queues can still grow indefinitely and cause "out of memory" errors. Our (stop-and-go) flow control mechanism acts whenever the size of the outgoing message queue is above a threshold. When this happens, we disable the generation of (most) outgoing messages as follows:

- We disable the generation of heartbeat messages (but the timeout for suspecting processes does not change!).
- We suspend the thread on the servers that receives client requests. This stops the generation of new outgoing messages. As a consequence, the UDP protocol which delivers client requests will start dropping requests, and this eventually slows down clients, for they have to retransmit the dropped requests.

Another queue where messages tend to accumulate is the queue holding unordered messages in the atomic broadcast algorithm: messages come in relatively fast, but go out relatively slowly, because deciding on their delivery order using consensus might take a lot of time. We limit the size of this queue using the same flow control mechanism as for the outgoing message queue.

9.5 How robust is our system?

The correctness of a distributed algorithm has two aspects: *safety* ("nothing bad ever happens") and *liveness* ("good things must eventually happen"). We call an algorithm *robust* if it is both safe and live, even when exposed to extremely high loads. The atomic broadcast algorithm that we chose [CT96] is safe under any conditions. Therefore, robustness is related to liveness in our experiment: is our atomic broadcast always able to deliver messages? The goal of our experiment is to find an answer to this question. The experiment has parameters which influence the load conditions of the system. For various settings of these parameters, we ran the experiment and checked whether the atomic broadcast algorithm was live. This section discusses the parameters of the experiment, as well as the method used for verifying liveness.

Note that we do not emulate process crashes in our experiment. This would primarily give information on the fault tolerance characteristics of the atomic broadcast algorithm, which are well understood [CT96]. The robustness of the algorithm is a major issue even if no crash occurs.[5]

[5] Note also that the FLP impossibility does not stem from the fact that crashes *do occur*, but from the fact that crashes *may happen* in an unanticipated manner at any point in the execution of the atomic broadcast algorithm, and that consequently, the algorithm has to be prepared for them.

9.5.1 Parameters of the experiment

We classify the parameters of our experiment into two categories: (1) *application parameters*, over which the implementor of the server has no control, and (2) *system parameters*, over which the implementor of the server has full control.

An application parameter influences the load on the network and the hosts. Our application parameter is r, the rate of requests coming from the clients, i.e., the number of requests per second.[6] A large r generates a high load on the network and on the replicated server. The number of clients is sufficiently high to maintain any reasonable value of r, even if the server processes requests very slowly. In order to demonstrate that our system is robust, we have to show that our replicated server works for any setting of the application parameter r.

Our system parameter is T, the timeout value for the failure detector. The time T_h between two consecutive heartbeat messages is set to $T/2$. Low timeout values yield frequent false suspicions, and thus increase the time needed to solve consensus, and for the client the time to get the reply after sending the request. High timeout values increase the detection time of the failure detector, and thus reaction time of the algorithm to process crashes.

As already mentioned in the introduction, the robustness of our server can easily be increased by setting T very high, say to one minute. However, this would imply that the replicated server may block for a minute when a process crashes.[7] We consider that such a behavior is unacceptable for a server replicated for high availability. For this reason, we explored how the replicated server behaves for small values of T.

9.5.2 Testing if the atomic broadcast algorithm can deliver messages

Given a setting of the parameters, how can we detect (1) if the atomic broadcast algorithm continues delivering messages forever or (2) if it will never deliver messages any more? The best that we can do is to detect conditions that allow us to conclude with some confidence that the behavior of the algorithm has stabilized. We use the following conditions to terminate a run of the experiment:

1. The clients have collected a certain number of replies (N) from the replicated server.

2. One instance of the consensus algorithm has not terminated after executing a certain number of rounds (R).

[6]Requests are generated by a Poisson process, thus we model independent requests. This is, however, not crucial to the experiment.

[7]Suppose that the coordinator of a round of the consensus algorithm crashes. At this moment, the other processes wait for either a message from the coordinator or that the failure detector starts suspecting the coordinator. With a timeout of one minute, the algorithm is blocked at this point for about one minute.

In every run of our experiment, one of these conditions is necessarily fulfilled. In case 1, we conclude that the algorithm was live in the current run: sending a request m using atomic broadcast eventually leads to the delivery of m, and thus to a reply to m. In case 2, the conclusion is that the algorithm was not live in the current run.

The values N and R should be chosen sufficiently high, to ensure that the behavior of the algorithm stabilizes. In the experiment, we used $N = 10\,000$ and $R = 10\,000$. $N = 10\,000$ was sufficient to ensure that each host participating in the experiment starts sending messages, and that at least 50 consensus are executed after the startup, for even the most unfavorable setting of the parameters r and T. $R = 10\,000$ was sufficient because we had no consensus that took 10 000 rounds: the consensus algorithm was always live.

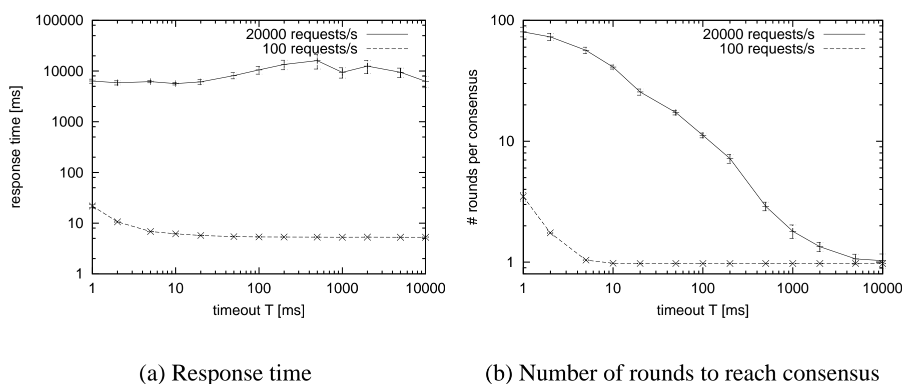
9.6 Results of our experiment

In spite of our expectations, we observed that the atomic broadcast algorithm works even under the most extreme conditions: a request rate that saturates the network (20 000 requests/s) and a very small detection time for the failure detector, approaching the resolution of the clock used (1 ms). We present the detailed results of the experiment in this section and discuss those results in Section 9.7.

We performed measurements for a variety of client request rates. We only present two sets of results which are characteristic: one for 100 requests/s and one for 20 000 requests/s (Figures 9.2(a) and 9.2(b)). The rate of 100 requests/s is well below the capacity of the replicated server. This rate corresponds to normal operation. Smaller rates give very similar results (with a different average response time). The other rate is 20 000 requests/s. It is pointless to increase the request rate beyond this point because at this rate, the network is already saturated with requests.[8] As for request rates between 100 requests/s and one for 20 000 requests/s, the observed behavior is in between the two extreme behaviors.

For both request rates and different timeout values, we measured two quantities: (1) the average response time (the time between the sending of a request and the reception of the corresponding reply, as seen by the client) and (2) the average number of rounds per consensus. The average response time is shown in Figure 9.2(a). The average number of rounds per consensus is shown in Figure 9.2(b). The characteristics of the "response time" curve and the "consensus rounds" curve are rather similar under moderate load; this is not surprising, as the number of rounds per consensus execution largely determines the response time. Under the extreme load, the response time is extremely high, and is influenced by

[8]The size of the Ethernet frame encapsulating a request is 383 bytes, and each request is sent in three copies (one per server). Knowing this and the bitrate of the Ethernet network (100 Mbit/s), we can compute an upper bound for the request rate: $\approx$ 11 500 requests/s (the real value is probably much lower, as the utilization of the network is usually far from 100%). Moreover, we measured that the each client host can pass at least 2000 requests/s through its socket interface. Thus 10 client hosts are certainly sufficient for overloading the network with requests.



(a) Response time (b) Number of rounds to reach consensus

Figure 9.2: Performance of the replicated server (three replicas) for an extreme and a moderate request rate r vs. the failure detection timeout T. Each point represents a mean value obtained from 100 independent experiments. The 95% confidence interval is shown.

other factors (queuing times in buffers and the flow control mechanisms described in Section 9.4.3).

In each curve (except for the response time curve at the extreme load) we can observe two kinds of behavior:

- at $r = 100/s$, the one behavior with a timeout value below $\overline{T} = 5\,\mathrm{ms}$, and the other behavior with a timeout value above $\overline{T}$.
- at $r = 20\,000/s$, the one behavior with a timeout value below $\overline{T} = 2000\,\mathrm{ms}$, and the other behavior with a timeout value above $\overline{T}$.

At high timeouts ($T \geq \overline{T}$), the measured quantities are largely independent of the timeout. The average number of consensus rounds is close to 1.

At low timeouts ($T < \overline{T}$), both the number of rounds and the response time (at moderate load) increase as the timeout decreases. This is due to the more and more frequent failure suspicions. Compared to the case of high timeouts, the confidence intervals for both the response time and the number of rounds are large, showing that these quantities are more unpredictable. We found that even at low timeouts, most consensus executions take relatively few rounds, but a few instances of consensus take a lot of rounds and thus increase the average significantly. The distribution of the number of rounds is shown in Fig. 9.3, for the most extreme setting of parameters: $r = 20\,000/s$ and $T = 1$ ms (as Fig. 9.2(b) shows, this setting of parameters results in the highest number of rounds per consensus execution on the average).

Finally, note that we did not try to optimize the response times of the server (shown in Fig. 9.2(a)). Even under light load (20 requests/s), the average response

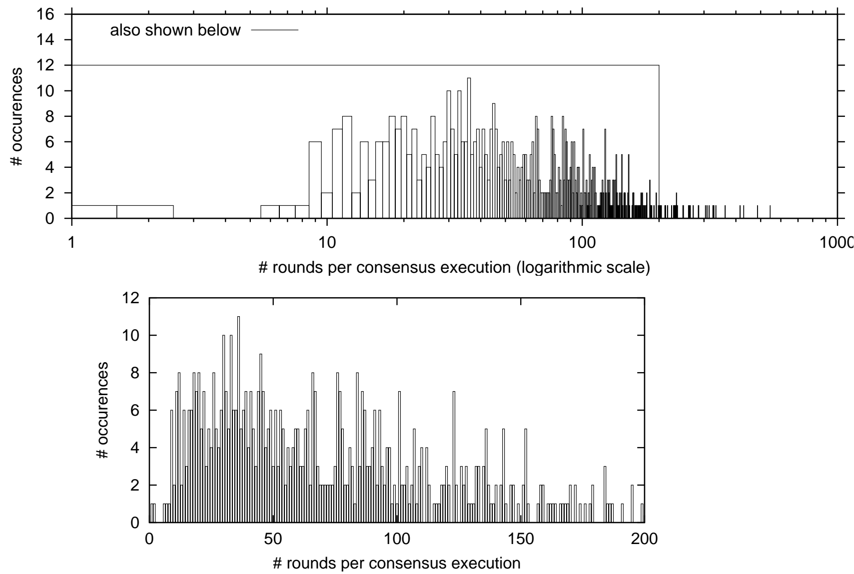


Figure 9.3: The distribution of the number of rounds per consensus execution, for $r = 20\,000/s$, $T = 1$ ms. 621 consensus executions are shown, coming from 10 independent experiments. The distribution of small round numbers is shown separately as well.

time was 4.36 ± 0.04 ms (for a sample of $10\,000$ independent requests).[9] Our goal was to evaluate the robustness of the server, and we could achieve this without optimizing the performance.

9.7 Discussion

The experiment showed that our replicated server is extremely robust. It worked under any conditions, even the most extreme ones: a request rate that saturates the network (20 000 requests/s) and a very small failure detection time, approaching the resolution of the clock used (1 ms). In this section, we discuss why it proved to be so robust.

The replicated server is robust because the underlying atomic broadcast algorithm is robust. In turn, as the atomic broadcast algorithm uses a sequence of consensus algorithms to decide what messages it can deliver next (Section 9.3), the atomic broadcast algorithm is robust because the underlying consensus algorithm always terminates. This can be explained as follows.

[9]An optimized implementation similar to the one used in the measurements described in could achieve a response time below 2 ms in this experiment. The main reason for the difference is that we could not use the efficient reliable broadcast algorithm of Section B.1, as this algorithm performs poorly if message losses and failure suspicions are frequent.

Recall from Section 9.3 that processes proceed in rounds in the consensus algorithm. In each round, a predetermined process acts as the coordinator. A successful round is a round in which a decision is taken. A round might fail because its coordinator may be suspected by other processes. Therefore the more often suspicions occur, the more rounds the consensus algorithm takes until it decides. The frequency of suspicions is directly related to the failure detection timeout T, and indirectly to the load of the system, influenced by the client request rate r. We now examine how the system behaves as we decrease T.

If T is high, consensus terminates in one round. This holds even if the system is loaded to the maximum extent, i.e., when client requests saturate the network ($r = 20\,000/s$). This is shown in Fig. 9.2(b), for $r = 100/s$ and $T \geq 20\,\mathrm{ms}$ and $r = 20\,000/s$ and $T \geq 500\,\mathrm{ms}$. The reason is that the coordinator is hardly ever suspected by the failure detector. This means that the coordinator can successfully send messages more often than at a frequency of $1/T$, as a failure detector stops suspecting a process whenever it receives a (recent enough) message from that process. This is not surprising: the Ethernet network strives to provide fair access to the transmission medium for each host on the network, and the result is that each host can successfully send a message every 500 ms.[10]

As we decrease T, suspicions get more and more frequent. With small timeout values, we expected that the coordinator of each round of the consensus algorithm would always be suspected, i.e., the consensus algorithm would forever proceed from one round to the next one without ever being able to decide. This is not the case: even for the smallest value for T and the highest possible load ($T = 1\,\mathrm{ms}$ and $r = 20\,000\,\mathrm{ms}$ in Fig. 9.2(b)) consensus executions take 81 rounds on the average, and the longest consensus execution we could find had 545 rounds. So, while consensus executions may take a large number of rounds and the number of rounds is rather unpredictable, each consensus execution terminates nevertheless. By analyzing logs of messages produced during the experiment, we were able to understand the reasons for this. We present our arguments in three steps:

1. The consensus algorithm tries to decide repeatedly, in every round. Therefore, if the algorithm does not terminate, the probability that a given round is successful (i.e., that the coordinator can decide in that round) must be very close to zero. We shall next argue that this is not the case.

2. Out of our three processes, one is always late: it never participates actively in the algorithm. The reason is that the algorithm needs the cooperation of only two processes (this is why it tolerates one crash failure). Thus the process that finishes one consensus execution late is likely to finish all subsequent executions late.

3. We now present a scenario depicting a successful round (see Fig. 9.4) and argue that this scenario occurs with a small but non-negligable probability. We

[10] Actually, Ethernet is the LAN technology that is the least fair among all LAN technologies. FDDI, for example, guarantees a bound on the access time to the shared medium.

only describe those details of the algorithm which are absolutely necessary for understanding the scenario.

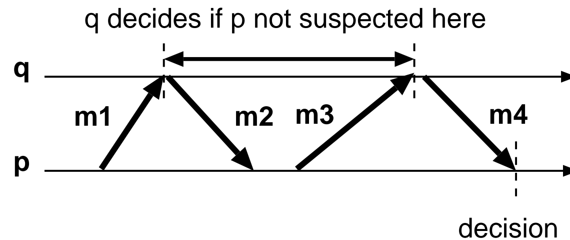


Figure 9.4: Consensus algorithm: q must suspect p before the reception of m_3 to prevent a decision in the current round. The late process is not shown.

(a) Process q is the coordinator of round r, and process p is the coordinator of round $r + 1$. The third process is the late process. This scenario likely repeats in every third round. The messages of the late process do not arrive in time to influence the scenario (and are thus omitted in Fig. 9.4).

(b) Process p sends m_1 to q in round r, indicating that round r failed. Upon receiving m_1, q starts executing round $r+1$, by sending a message m_2 to the new coordinator p. Receiving m_1 also indicated to q that p is alive. Therefore q likely will not suspect p for some time after sending m_2.

(c) Process p waits for m_2. Upon receiving m_2, it sends p_3 to q. In the meantime, process q waits either (1) until it receives message m_3 from p, or (2) until it suspects p.

(d) *If q receives m_3 before suspecting p, then p will be able to decide* (when receiving the reply m_4). Messages of the algorithm reset the timer of the heartbeat failure detector, hence it is unlikely that q will *always* suspect p between the reception of m_1 and the reception of m_3. Consequently, in every third round (at least), the decision may take place. Thus eventually, there is one round in which the coordinator decides, and forces the other processes to decide.

Our lowest setting for the failure detection time T_D was 1 ms. The question arises whether we could have lowered the timeout value and observe consensus executions that do not terminate. The answer is that we could have lowered T_D, and possibly observed consensus executions which do not terminate, but such small T_D values do not make sense in practice. The reason is the following. The motivation to decrease T_D is to speed up the reaction of the algorithm to crashes. The reaction time is difficult to define precisely, but it is certainly related to the performance of the replicated server from the client's point of view. Decreasing T further can

improve the response time of the server by at most 1 ms in the case of a crash. This improvement is insignificant: recall from Section 9.6 that the best case (no crashes, no suspicions, light load) response time of our server is 4.36 ms.

Chapter 10

Conclusion

10.1 Research assessment

The thesis has major contributions in three domains. As for new tools, a simulation and prototyping framework offers a practical and some new complexity metrics a theoretical tool for the performance evaluation of agreement algorithms. As for methodology, the thesis proposed a set of well-defined benchmarks for atomic broadcast algorithms; such algorithms are important as they provide the basis for a number of replication techniques. Finally, three case studies were presented that investigate important performance issues with agreement algorithms. Let us now assess each of the contributions in more detail.

The Neko prototyping and simulation framework. We presented Neko, a simple Java communication platform that provides support for simulating and prototyping distributed algorithms. The same implementation of the algorithm can be used both for simulations and executions on a real network; this reduces the time needed for performance evaluation and thus the overall development time. Neko is also a convenient implementation platform which does not incur a major overhead on communications. Neko is written in Java and is thus highly portable. It was deliberately kept simple, easy to use and extensible: e.g., some more types of real or simulated networks could be added or integrated easily. It includes a library of agreement algorithms, as well as support for centralized configuration, the control of experiments and gathering statistics. It was successfully used to conduct all the simulation and measurement studies in this thesis, and was also used in a number of student projects at EPFL.

Contention-aware performance metrics. We proposed two metrics to predict the latency and the throughput of distributed algorithms. Unlike other existing metrics, the two complementary metrics that we present here take account of both network and CPU contention. This allows for more accurate predictions and a finer grained analysis of algorithms than what time complexity and message complexity

permit; we have shown this using analytical computations for the throughput metric and both analytical computations and measurements for the latency metric. In addition, our metrics make it possible to find out whether the bottleneck is the network or the CPU of one specific process.

The problem of resource contention is commonly recognized as having a major impact on the performance of distributed algorithms. Because other metrics do not take account of contention to the same extent as ours, our metrics fill a gap between simple complexity measures and more complex performance evaluation techniques.

Beside the metrics themselves, a tool that helps evaluating them was presented, and extensive validation studies were conducted. The use of the metrics was also illustrated in the two chapters describing detailed simulation studies.

Benchmarks for atomic broadcast algorithms. In the field of group communication, comparing performance results from different sources is difficult, partly due to the lack of well-defined benchmarks. In an attempt to consolidate the situation, we defined a number of benchmarks for atomic broadcast, an important agreement problem requiring that all destinations of messages deliver the same set of messages in the same order. The benchmarks include well-defined metrics, workloads and failure scenarios (faultloads). The use of the benchmarks was illustrated in the two chapters describing detailed simulation studies. We hope that other researchers will find the simplicity and implementation independent formulation of the benchmarks attractive enough to start using them.

Comparing unreliable failure detectors and group membership. We compared the performance of two atomic broadcast algorithms designed for an asynchronous system. The two algorithms use different mechanisms for handling failures: unreliable failure detectors which provide *inconsistent* information about failures, and a group membership service which provides *consistent* information about failures, respectively.

Based on our results, we advocate a combined use of the two approaches to failure handling: unreliable failure detectors should be used for making failure handling more responsive and robust; and a group membership service with a differently tuned failure detector, making fewer mistakes, should be used to get the long term resiliency and performance benefits after crashes occur.

Comparing centralized and decentralized coordination. The second of our detailed case studies compared two consensus algorithms designed for an asynchronous system. They differ in how they coordinate the decision process: the one uses a centralized and the other a decentralized communication schema. It must be noted that the problem of centralized and decentralized communication recurs in solutions to a lot of agreement problems.

Our results show that the relative performance of the two algorithms is highly affected by a number of characteristics of the environment, like the availability of multicast and the amount of contention on the hosts versus the amount of contention on the network. This information can be used to guide deployment decisions.

On the robustness of replicated servers. Famous theoretical results state that consensus, atomic broadcast and group membership are not solvable in the asynchronous system model. However, the impact of these results to implementations of replicated services is unclear.

We investigated the problem by stress testing a replicated service in a local area network. As the impossibility results are based on the difficulty of distinguishing slow and crashed processes, we put a focus on the failure detectors underlying the replicated service. In particular, we overloaded the network with requests, and required that the failure detectors detect crashes very fast (as the impossibility results stem from the difficulty of doing reliable failure detection). We found that our replicated server continued working even with the most extreme settings. Thus, by trying to investigate the effects of impossibility results, we discovered how to implement replicated servers which are extremely robust in a local area network setting.

10.2 Open questions and future research directions

Further development on the Neko framework. We plan to continue developing Neko. The short term goals include implementing some more components useful for group communication, and integrating a transport layer with an efficient (IP multicast based) reliable multicast protocol. Some long term goals are integration with an advanced network simulator (such as NS-2 [FK00]), and improving the efficiency of message serialization (currently, the standard Java serialization mechanisms are used). Also, we plan to switch to XML based configuration files, to have a cleaner way of configuring parts of a Neko application (i.e., configuration entries with a scope). Furthermore, there is a need for more sophisticated protocol composition features than the simple protocol stacking currently used. The most ambitious project involves re-working the internals of Neko to make it suitable for studies of scalability, with hundreds or thousands of nodes.

The Neko source code is available freely at http://lsrwww.epfl.ch/neko [Urb00], along with documentation. Given sufficient interest, we will set up an Open Source project around it.

Contention-aware performance metrics. The system model for the contention-aware metrics (including one CPU resource per host and a shared network resource can be extended in a variety of ways. For example, modeling a separate network

processor beside the CPUs would bring it closer to the architecture of current networks. Also, the bus-like network could be replaced by more complex topologies.[1] Careful experimentation is needed to decide which extensions result in more realistic models, without making the computation of the metrics unnecessarily difficult. Note, in particular, that a principal merit of the contention-aware metrics is that there is only one parameter.

Benchmarks for atomic broadcast algorithms. Future work will probably require extensions to the set of benchmarks. Asymmetric workloads and bursty workloads are likely needed, as there are atomic broadcast algorithms optimized for such workloads. Self-similar workloads hold a promise for modeling burstiness. If the algorithm being analyzed tolerates message losses and/or Byzantine process failures beside crash failures of processes, such failures should be included in the set of faultloads.

Similar benchmarks could be defined for other agreement problems. Adapting the benchmarks to other multicast primitives, with different ordering guarantees, is relatively easy. Many-to-many communication primitives, like consensus, are more problematic.

Comparing unreliable failure detectors and group membership. We have chosen atomic broadcast algorithms with a centralized communication scheme, with one process coordinating the others. The algorithms are practical: in the absence of crashes and suspicions, they are optimized to have small latency under low load, and to work under high load as well (messages needed to establish delivery order are aggregated). In the future, we would like to investigate algorithms with a decentralized communication scheme (e.g., [Lam78, AS00]) as well.

Another future research direction is to compare non-uniform atomic broadcast algorithms (our study was performed with uniform algorithms). Some theoretical results give hints that failure detector based algorithms cannot take advantage of non-uniformity to decrease their latency. In contrast, the group membership based algorithm has an efficient non-uniform variant, though we must point out that, unlike in our study, a state transfer to wrongly excluded processes (as opposed to simply replaying the messages missed) cannot be avoided when using the non-uniform version of the algorithm. Any studies on this issue should include the cost of state transfer into their model.

Comparing centralized and decentralized coordination. In our study, we assumed that all hosts work at equal speeds. A real system often has more asymmetric configurations, either because the hardware is different or the hosts are loaded to a different extent. The behavior of the algorithms should be investigated in such

[1] We view the bus-like network of the model as a mechanism to introduce network contention, rather than an attempt at modeling Ethernet networks. In fact, it is the simplest such mechanism we could think of.

systems. The algorithms could even be modified to adapt to asymmetric configurations and changing loads. E.g., each consensus execution could decide which process should be the first coordinator for the next consensus execution. The first coordinator should preferably be on a lightly loaded or relatively powerful host.

On the robustness of replicated servers. Our experiment was performed in a local area network (LAN) setting. It would be interesting to perform a similar experiment in a Wide Area Network (WAN) as well. In such a setting, message delays vary a lot more than in a LAN, partly because message losses are much more frequent. Hosts can even be partitioned from each other for a long time. Such an environment is definitely closer than a LAN to the asynchronous model in which the impossibility results about various agreement problems were proven, and an experiment might unveil their effects.

Bibliography

[AADW94] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 35th Annual Symp. on Foundations of Computer Science*, pages 401–411, Los Alamitos, CA, USA, November 1994.

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. 12th Int'l Symp. on Distributed Computing (DISC)*, pages 231–245, Andros, Greece, September 1998.

[ACT99] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999. Special issue on distributed algorithms.

[AKP92] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proc. 24th ACM Symp. on Theory of Computing (STOC)*, pages 571–580, Victoria, BC, Canada, May 1992.

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, November 1995.

[AS00] M. K. Aguilera and R. E. Strom. Efficient atomic broadcast using deterministic merge. In *Proc. 19th Annual ACM Int'l Symp. on Principles of Distributed Computing (PODC-19)*, pages 209–218, Portland, OR, USA, July 2000.

[AW95] J. Aspnes and O. Waarts. A modular measure of competitiveness for distributed algorithms (abstract). In *Proc. 14th ACM Symp. on Principles of Distributed Computing (PODC-14)*, page 252, Ottawa, Ontario, Canada, August 1995.

[AW96] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proc. 28th ACM Symp. on Theory of Computing (STOC)*, pages 237–246, Philadelphia, PA, USA, May 1996.

[BC94] K. P. Birman and T. Clark. Performance of the ISIS distributed computing toolkit. Technical Report TR94-1432, Cornell University, Computer Science Department, June 1994.

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

[BCFK99] M. Baker, B. Carpenter, G. Fox, and S. H. Koo. mpiJava: An object-oriented Java interface to MPI. *Lecture Notes in Computer Science*, 1586:748–762, 1999.

[BFR92] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proc. 24th ACM Symp. on Theory of Computing (STOC)*, pages 39–50, Victoria, BC, Canada, May 1992.

[BHSC98] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–366, November 1998.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in presence of failures. *ACM Trans. on Computer Systems*, 5(1):47–76, February 1987.

[BMD93] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in distributed computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

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

[BS96] A. Beguelin and V. Sunderam. Tools for monitoring, debugging, and programming in PVM. In *Proc. 3rd European PVM Conf. (EuroPVM'96)*, pages 7–13, Munich, Germany, October 1996.

[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.

[BvR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[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 *Proc.*

20th IEEE Symp. on Reliable Distributed Systems (SRDS), pages 244–249, Osaka, Japan, October 2002.

[CBG01] A. Coccoli, A. Bondavalli, and F. D. Giandomenico. Analysis and estimation of the quality of service of group communication protocols. In *Proc. 4th IEEE Int'l Symp. on Object-oriented Real-time Distributed Computing (ISORC'01)*, pages 209–216, Magdeburg, Germany, May 2001.

[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.

[CdBM94] F. Cristian, R. de Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering Journal*, 1(4):177–201, June 1994.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. on Parallel & Distributed Systems*, 10(6):642–657, June 1999.

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

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

[CKP+96] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Comm. ACM*, 39(11):78–85, November 1996.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, May 2001.

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

[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, February 1991.

[CSG+00] A. Coccoli, S. Schemmer, F. D. Giandomenico, M. Mock, and A. Bondavalli. Analysis of group communication protocols to assess quality of service properties. In *Proc. IEEE High Assurance System Engineering Symp. (HASE'00)*, pages 247–256, Albuquerque, NM, USA, November 2000.

[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 Trans. 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 Performance and Dependability Symp.*, pages 551–560, Washington, DC, USA, June 2002.

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

[DAK00] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal*, 39(1), 2000.

[DCS97] H. Duggal, M. Cukier, and W. Sanders. Probabilistic verification of a synchronous round-based consensus protocol. In *Proc. 16th Symp. on Reliable Distributed Systems (SRDS '97)*, pages 165–174, Durham, North Carolina, USA, October 1997.

[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.

[Dee89] S. E. Deering. RFC 1112: Host extensions for IP multicasting, August 1989.

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

[DHW97] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of ACM*, 44(6):779–805, November 1997.

[Din99] K. Dincer. A ubiquitous message passing interface: jmpi. In *Proc. 13th Int'l Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing*, pages 203–207, San Juan, Puerto Rico, USA, April 1999.

[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.

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

[DSU00] X. Défago, A. Schiper, and P. Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Technical Report DSC-2000-036, École Polytechnique Fédérale de Lausanne, Switzerland, September 2000.

[DSYB90] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon. NEST: A network simulation and prototyping testbed. *Comm. ACM*, 33(10):63–74, October 1990.

[ECM02] C. E. Campbell, Jr. and T. McRoberts. *The Simple Sockets Library*, 2002.

[EMS95] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proc. 15th Int'l Conf. on Distributed Computing Systems (ICDCS-15)*, pages 296–306, Vancouver, Canada, May 1995.

[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, May 2003.

[Fer98] A. Ferrari. JPVM: network parallel computing in Java. *Concurrency: Practice and Experience*, 10(11–13):985–992, September 1998.

[FGS97] P. Felber, R. Guerraoui, and A. Schiper. Replicating objects using the CORBA event service. In *Proc. 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 14–19, Tunis, Tunisia, October 1997.

[FK00] K. Fall and Kannan Varadhan, editors. *The ns Manual*, 2000.

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

[FP01] S. Frolund and F. Pedone. Revisiting reliable broadcast. Technical Report HPL-2001-192, HP Laboratories, Palo Alto, CA, USA, August 2001.

[FvR97] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE Symp. on High Performance Distributed Computing*, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Publishing Company, Reading, MA, 1995.

[GHLL+98] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference. Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, USA, second edition, 1998. See also volume 1 [SOHL+98].

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

[Gra86] J. Gray. Why do computers stop and what can be done about it ? In *Proc. 5th Symp. on Reliablity in Distributed Software and Database systems*, January 1986.

[GS93] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Texts and monographs in computer science. Springer-Verlag, 1993.

[GS01] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. on Software Engineering*, 27(1):29–41, January 2001.

[Gue95] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proc. 9th Int'l Workshop on Distributed Algorithms (WDAG-9)*, pages 87–100, Le Mont-St-Michel, France, September 1995.

[Hay98] M. Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, Computer Science, January 1998.

[HLL98] J.-F. Hermant and G. Le Lann. A protocol and correctness proofs for real-time high-performance broadcast networks. In *Proc. 18th Int'l Conf. on Distributed Computing Systems (ICDCS-18)*, pages 360–369, Amsterdam, The Netherlands, May 1998.

[HM98] F. Howell and R. McNab. SimJava: a discrete event simulation package for Java with applications in computer systems modelling. In *Proc. First Int'l Conf. on Web-based Modelling and Simulation*, San Diego, CA, USA, January 1998.

[HP91] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Trans. Software Engineering*, 17(1):64–76, January 1991.

[HP97] A. Heddaya and K. Park. Congestion control for asynchronous parallel computing on workstation networks. *Parallel Computing*, 23(13):1855–1875, December 1997.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. Addison Wesley, second edition, 1993.

[HUSK02] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama. Performance comparison between the Paxos and Chandra-Toueg consensus algorithms. In *Proc. Int'l Arab Conf. on Information Technology (ACIT 2002)*, pages 526–533, Doha, Qatar, December 2002.

[HYF99] J.-H. Huang, C.-C. Yang, and N.-C. Fang. A novel congestion control mechanism for multicast real-time connections. *Computer Communications*, 22:56–72, 1999.

[IBM00] IBM Corporation. *SockPerf: A Peer-to-Peer Socket Benchmark Used for Comparing and Measuring Java Socket Performance*, 2000.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, New York, NY, USA, May 1991.

[Kri96] E. V. Krishnamurthy. Complexity issues in parallel and distributed computing. In A. Y. H. Zomaya, editor, *Parallel & Distributed Computing Handbook*, pages 89–126. McGraw-Hill, 1996.

[Kro96] L. I. Kronsjö. PRAM models. In A. Y. H. Zomaya, editor, *Parallel & Distributed Computing Handbook*, pages 163–191. McGraw-Hill, 1996.

[KT91a] F. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proc. 11th Int'l Conf. on Distributed Computing Systems (ICDCS-11)*, pages 222–230, Arlington, TX, USA, May 1991.

[KT91b] M. F. Kaashoek and A. S. Tanenbaum. Fault tolerance using group communication. *ACM Operating Systems Review*, 25(2):71–74, April 1991.

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

[Lam01] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):34–58, December 2001.

[LTWW94] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ ACM Trans. on Networking*, 2(1):1–15, February 1994.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[LYZ91] C.-C. Lim, L.-J. Yao, and W. Zhao. A comparative study of three token ring protocols for real-time communications. In *Proc. 11th Int'l Conf. on Distributed Computing Systems (ICDCS-11)*, pages 308–317, May 1991.

[May92] E. Mayer. An evaluation framework for multicast ordering protocols. In *Proc. Conf. on Applications, Technologies, Architecture, and Protocols for Computer Communication (SIGCOMM)*, pages 177–187, August 1992.

[MDB01] A. Montresor, R. Davoli, and Ö. Babaoğlu. Middleware for dependable network services in partitionable distributed systems. *Operating Systems Review*, 35(1):73–84, January 2001.

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

[MKA+01] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johansson, R. Lindström, et al. Preliminarily dependability benchmark framework. Project deliverable CF2, Dependability Benchmarking project (DBench), EC IST-2000-25425, August 2001.

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

[MPR01] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st Int'l Conf. on Distributed Computing Systems (ICDCS' 01)*, pages 707–710, Washington - Brussels - Tokyo, April 2001.

[MR99] A. Mostéfaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proc. 13th Int'l Symp. on Distributed Computing (DISC)*,

number 1693, pages 49–63, Bratislava, Slovak Republic, September 1999.

[MS95] C. P. Malloth and A. Schiper. View synchronous communication in large scale distributed systems. In *Proc. 2nd Open Workshop of ESPRIT project BROADCAST (6360)*, Grenoble, France, July 1995.

[MSS95] L. M. Malhis, W. H. Sanders, and R. D. Schlichting. Numerical evaluation of a group-oriented multicast protocol using stochastic activity networks. In *Proc. 6th Int'l Workshop on Petri Nets and Performance Models*, pages 63–72, Durham, NC, USA, October 1995.

[Neu03] P. G. Neumann. The risk digest: Forum on risks to the public in computers and related systems. *ACM Committee on Computers and Public Policy*, 22(75), 2003.

[PBS89] L. L. Peterson, N. C. Bucholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. on Computer Systems*, 7(3):217–246, 1989.

[PHN00] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[Pol94] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994.

[PSUC02] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proc. 4th European Dependable Computing Conf. (EDCC-4)*, volume 2485 of *LNCS*, pages 44–61, Toulouse, France, October 2002. Springer-Verlag.

[Ray88] M. Raynal. *Networks and Distributed Computation: concepts, tools, and algorithms*. MIT Press, 1988.

[RPL+02] H. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. Sanders. Quantifying the cost of providing intrusion tolerance in group communication systems. In *Proc. 2002 Int'l Conf. on Dependable Systems and Networks (DSN-2002)*, pages 229–238, Washington, DC, USA, June 2002.

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

[Sch93] F. B. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 7, pages 169–198. Addison-Wesley, second edition, 1993.

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

[SDP91] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, pages 66–73, 1991.

[SDS01] N. Sergent, X. Défago, and A. Schiper. Impact of a failure detection mechanism on the performance of consensus. In *Proc. IEEE Pacific Rim Symp. on Dependable Computing (PRDC)*, pages 137–145, Seoul, Korea, December 2001.

[Ser98] N. Sergent. *Soft Real-Time Analysis of Asynchronous Agreement Algorithms Using Petri Nets*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1808.

[SH99] R. Schlichting and M. Hiltunen. The Cactus project, 1999.

[SOHL+98] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference. Volume 1, The MPI-1 Core*. MIT Press, Cambridge, MA, USA, second edition, September 1998. See also volume 2 [GHLL+98].

[ST85] D. D. Sleator and R. E. Tarjan. Amortised efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, February 1985.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, December 1990.

[TBW95] K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, September 1995.

[THIS97] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An operating system coordinated high performance communication library. In *Proc. High-Performance Computing and Networking (HPCN'97 Europe)*, Vienna, April 1997.

[UDS00] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics: Analysis of distributed algorithms. Technical Report DSC-2000-012, École Polytechnique Fédérale de Lausanne, Switzerland, February 2000.

[Urb00] P. Urbán. *The Neko web pages*. École Polytechnique Fédérale de Lausanne, Switzerland, 2000.

[Val90] L. G. Valiant. A bridging model for parallel architectures. *Comm. ACM*, 33(8):103–111, August 1990.

[VBH+98] R. Van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software: Practice and Experience*, 28(9):963–979, July 1998.

[VRBM96] R. Van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Comm. ACM*, 39(4):76–83, April 1996.

[Wie02] M. Wiesmann. *Group Communications and Database Replication: Techniques, Issues and Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, May 2002. Number 2577.

[WMS02] P. Wojciechowski, S. Mena, and A. Schiper. Semantics of protocol modules composition and interaction. In *5th Int'l Conf. COORDINATION 2002*, pages 389–404, York, UK, April 2002.

Appendix A

Atomic broadcast algorithms

All atomic broadcast algorithms used in this thesis have been informally described in Section 5.6. In this appendix, we first introduce the pseudocode notation in Section A.1. We then present the assumptions on which the algorithms are based, and we give the pseudocode for each algorithm, in Section A.2. Finally, Section A.3 gives the full formulas obtained by the analysis for the contention-aware metrics. Most pseudocode descriptions and formulas are taken from [Déf00] with small adaptations only.

A.1 Pseudocode notation

The pesudocode notation is rather straightforward. A few elements, however, deserve some explanation.

Parallelism. Most algorithms are described in an event triggered manner: **procedure**s are called and **when** statements are triggered when their condition becomes true. Only one **when** statement or **procedure** executes at any given time, unless execution is blocked in a **wait until** statement or new tasks are **spawn**ed explicitly. This policy helps avoiding problems arising from parallel access to variables.

Sequences. A lot of algorithms are expressed with sequences, that is, finite ordered lists of elements. With a few minor exceptions, the notation defined here is borrowed from Gries and Schneider [GS93].

A sequence of elements a, b, and c is denoted by the tuple $\langle a, b, c\rangle$. The symbol ϵ denotes the empty sequence.

Elements can be added at the end of the sequence. Adding an element e at the end of a sequence seq is called appending and is denoted by $seq \triangleright e$. The same operator is used for appending sequences.

The first element of the sequence is denoted as $head.seq$. The tail of a non-empty sequence seq is the sequence that results from removing the first element

of seq. Thus, we have

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

Sequences are also considered sets and are used with the standard set notation: e.g., given an element e and a sequence seq, e is an element of seq (denoted $e \in seq$) if e is an element of the set composed of all elements of seq. The difference of a sequence seq and a set S is worth mentioning: $seq \setminus S$ produces the sequence that results from removing from seq all elements that appear in S, thereby keeping the order of elements in seq.

Sending messages to all. Unless stated otherwise, when a process p sends a message to all ("send m to all"), it sends that message to all processes *except* itself. When it is *explicitly* stated that a process p sends a message to all *including itself* ("send m to all (including itself)"), the copy of the message that is sent to itself does not actually transit on the network. The system only activates the right **when** statement to handle the message.

A.2 Pseudocode descriptions

The algorithms are based on the following assumptions (see Chapter 2 for the definitions of the terms used):

- There is no bound on communication delay or process speed (asynchronous system).
- The algorithms can be made tolerant of process crashes using a group membership service.
- Communication channels are reliable (implies no partition).

Exceptions are marked when appropriate.

A.2.1 Non-uniform sequencer algorithm

Code of sequencer:
  Initialization:
    $seqnum \leftarrow 0$ *{last seq. number attributed to a message}*

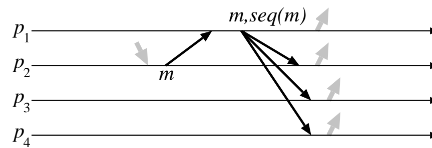
  procedure *A-broadcast*(m) *{To A-broadcast a message m}*
    $seqnum \leftarrow seqnum + 1$
    send $(m, seqnum)$ to all
    deliver (m)

  when receive (m)
    A-broadcast(m)

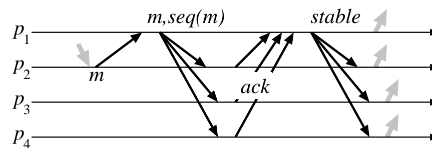
Code of all processes except sequencer:
  Initialization:
    $lastdelivered_p \leftarrow 0$ *{sequence number of the last delivered message}*
    $received_p \leftarrow \emptyset$ *{set of received yet undelivered messages}*

  procedure *A-broadcast*(m) *{To A-broadcast a message m}*
    send (m) to sequencer

  when receive $(m, seq(m))$
    $received_p \leftarrow received_p \cup \{(m, seq(m))\}$
    while $\exists m', seq$ s.t. $(m', seq) \in received_p \wedge seq = lastdelivered_p + 1$ **do**
      deliver (m')
      $lastdelivered_p \leftarrow lastdelivered_p + 1$
      $received_p \leftarrow received_p \setminus \{(m', seq)\}$



A.2.2 Uniform sequencer algorithm



Code of sequencer:

Initialization:

$seqnum \leftarrow 0$ *{last seq. number attributed to a message}*
$lastdelivered \leftarrow 0$ *{sequence number of the last delivered message}*
$received \leftarrow \emptyset$ *{set of received messages}*
$stable \leftarrow \emptyset$ *{set of stable yet undelivered messages}*

procedure *A-broadcast*(m) *{To A-broadcast a message m}*
$seqnum \leftarrow seqnum + 1$
send $(m, seqnum)$ to all
$received \leftarrow received \cup \{(m, seqnum)\}$
spawn
wait until $\forall q \in \pi(t)$: received (m, seq, ack)
send $(m, seq, stable)$ to all
$stable \leftarrow stable \cup \{(m, seq(m))\}$
while $\exists (m', seq') \in stable$ s.t. $seq' = lastdelivered + 1$ **do**
deliver (m')
$lastdelivered \leftarrow lastdelivered + 1$
$stable \leftarrow stable \setminus \{(m', seq')\}$

when receive (m)
A-broadcast(m)

Code of all processes except sequencer:

Initialization:

$lastdelivered_p \leftarrow 0$ *{sequence number of the last delivered message}*
$received_p \leftarrow \emptyset$ *{set of received messages}*
$stable_p \leftarrow \emptyset$ *{set of stable yet undelivered messages}*

procedure *A-broadcast*(m) *{To A-broadcast a message m}*
send (m) to sequencer

when receive (m, seq)
$received_p \leftarrow received_p \cup \{(m, seq)\}$
send (m, seq, ack) to sequencer

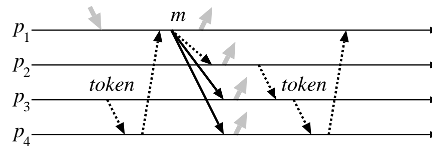
when receive $(m, seq, stable)$
$stable_p \leftarrow stable_p \cup \{(m, seq)\}$
while $\exists (m', seq') \in stable_p$ s.t. $seq' = lastdelivered_p + 1$ **do**
deliver (m')
$lastdelivered_p \leftarrow lastdelivered_p + 1$
$stable_p \leftarrow stable_p \setminus \{(m', seq')\}$

A.2.3 Non-uniform privilege-based algorithm

Initialization:
  $sendQ_p \leftarrow \epsilon$ *{sequence of messages to send (send queue)}*
  $recvQ_p \leftarrow \epsilon$ *{sequence of received messages (receive queue)}*
  $lastdelivered_p \leftarrow 0$ *{sequence number of the last delivered message}*
  $toknext_p \leftarrow p + 1 (\mathrm{mod}\, n)$ *{identity of the next process along the logical ring}*
  if $p = p_1$ **then** *{virtual message to initiate the token rotation}*
    send $(\perp, 0, 1)$ to p_1 *{format: (message, seq. number, next token holder)}*

procedure *A-broadcast*(m) *{To A-broadcast a message m}*
  $sendQ_p \leftarrow sendQ_p \rhd m$

when receive $(m, seqnum, tokenholder)$
  if $m \neq \perp$ **then** *{Receive new messages}*
    $recvQ_p \leftarrow recvQ_p \rhd (m, seqnum)$
  if $p = tokenholder$ **then** *{Circulate token, if appropriate}*
    if $sendQ_p \neq \epsilon$ **then** *{Send pending messages, if any}*
      $msg \leftarrow head.sendQ_p$
      $sendQ_p \leftarrow tail.sendQ_p$
      send $(msg, seqnum + 1, toknext_p)$ to all
      $recvQ_p \leftarrow recvQ_p \rhd (msg, seqnum + 1)$
    else
      send $(\perp, seqnum, toknext_p)$ to $toknext_p$
  while $\exists m'$ s.t. $(m', lastdelivered_p + 1) \in recvQ_p$ **do** *{Deliver messages that can be}*
    $recvQ_p \leftarrow recvQ_p \setminus \{m'\}$
    deliver (m')
    $lastdelivered_p \leftarrow lastdelivered_p + 1$

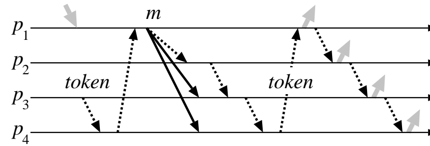


A.2.4 Uniform privilege-based algorithm

Initialization:
  $sendQ_p \leftarrow \epsilon$ *{sequence of messages to send (send queue)}*
  $recvQ_p \leftarrow \epsilon$ *{sequence of received messages (receive queue)}*
  $stableQ_p \leftarrow \epsilon$ *{sequence of stable messages (stable queue)}*
  $lastdelivered_p \leftarrow 0$ *{sequence number of the last delivered message}*
  $toknext_p \leftarrow p + 1 (\mathrm{mod} n)$ *{identity of the next process along the logical ring}*
  $acks_p \leftarrow (\emptyset, \ldots, \emptyset)$ *{array $[p_1, \ldots, p_n]$ of message sets (acknowledged messages)}*
  if $p = p_1$ **then** *{send a virtual message to initiate the token rotation}*
    send $(\bot, 0, 1, acks_p)$ to p_1 *{format: (message, seq. number, next token holder, acks)}*

procedure *A-broadcast*(m) *{To A-broadcast a message m}*
  $sendQ_p \leftarrow sendQ_p \triangleright m$

when receive $(m, seqnum, tokenholder, acks)$
  if $m \neq \bot$ **then** *{Receive new messages}*
    $recvQ_p \leftarrow recvQ_p \triangleright (m, seqnum)$
  while $\exists m'$ s.t. $(m', seq') \in recvQ_p$ **do** *{Ack recv'd messages and detect stability}*
    $acks[p] \leftarrow acks[p] \cup \{m'\}$
    if $\forall q \in \pi(t) : m' \in acks[q]$ **then**
      $stableQ_p \leftarrow stableQ_p \triangleright (m', seq')$
      $recvQ_p \leftarrow recvQ_p \setminus \{(m', seq')\}$
  if $p = tokenholder$ **then** *{Circulate token, if appropriate}*
    if $sendQ_p \neq \epsilon$ **then** *{Send pending messages, if any}*
      $msg \leftarrow head.sendQ_p$
      $sendQ_p \leftarrow tail.sendQ_p$
      send $(msg, seqnum + 1, toknext_p, acks)$ to all
      $recvQ_p \leftarrow recvQ_p \triangleright (msg, seqnum + 1)$
    else
      send $(\bot, seqnum, toknext_p, acks)$ to $toknext_p$
  while $\exists m'$ s.t. $(m', lastdelivered_p + 1) \in stableQ_p$ **do** *{Deliver messages that can be}*
    $stableQ_p \leftarrow stableQ_p \setminus \{m'\}$
    deliver (m')
    $lastdelivered_p \leftarrow lastdelivered_p + 1$



A.2.5 Uniform communication history algorithm

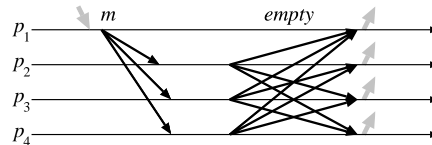
The channels used by this algorithm are reliable and FIFO.

Initialization:
  $received_p \leftarrow \emptyset$ *{Messages received by process p}*
  $delivered_p \leftarrow \emptyset$ *{Messages delivered by process p}*
  $deliverable_p \leftarrow \emptyset$ *{Messages ready to be delivered by process p}*
  $LC_p[p_1, \ldots, p_n] \leftarrow \{0, \ldots, 0\}$ *{ $LC_p[q]$: logical clock of process q as seen by process p}*

procedure *A-broadcast*(m) *{To A-broadcast a message m}*
  $LC_p[p] \leftarrow LC_p[p] + 1$
  send $(m, LC_p[p])$ to all

when no message sent for Δ_{live} time units
  $LC_p[p] \leftarrow LC_p[p] + 1$
  send $(\perp, LC_p[p])$ to all

when receive $(m, ts(m))$
  $LC_p[p] \leftarrow \max(ts(m), LC_p[p]) + 1$ *{Update logical clock}*
  $LC_p[sender(m)] \leftarrow ts(m)$
  $received_p \leftarrow received_p \cup \{m\}$
  $deliverable_p \leftarrow \emptyset$
  for each message m' in $received_p \setminus delivered_p$ **do**
    if $ts(m') < \min_{q \in \pi(t)} LC_p[q]$ **then**
      $deliverable_p \leftarrow deliverable_p \cup \{m'\}$
  deliver all messages in $deliverable_p$, according to the total order $\Longrightarrow$ (see Sect. 5.6.3)
  $delivered_p \leftarrow delivered_p \cup deliverable_p$



A.2.6 Non-uniform destinations agreement algorithm

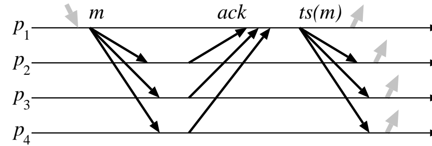
Initialization:
  $\mathit{received}_p \leftarrow \emptyset$ *{set of messages received by process p, with a temporary local timestamp}*
  $\mathit{stamped}_p \leftarrow \emptyset$ *{set of messages received by process p, with a final global timestamp}*
  $LC_p \leftarrow 0$ *{LC_p: logical clock of process p}*

procedure *A-broadcast*(m) *{To A-broadcast a message m}*
  send $(m, \mathit{nodeliver})$ to all (including itself)

  spawn
    wait until $\forall q \in \pi(t)$: received $(m, ts_q(m))$
    $TS(m) \leftarrow \max_{q\in\pi(t)} ts_q(m)$
    send $(m, TS(m), \mathit{deliver})$ to all (including itself)

when receive $(m, \mathit{nodeliver})$
  $ts_p(m) \leftarrow LC_p$
  $\mathit{received}_p \leftarrow \mathit{received}_p \cup \{(m, ts_p(m))\}$
  send $(m, ts_p(m))$ to $\mathit{sender}(m)$
  $LC_p \leftarrow LC_p + 1$

when receive $(m, TS(m), \mathit{deliver})$
  $\mathit{stamped}_p \leftarrow \mathit{stamped}_p \cup \{(m, TS(m))\}$
  $\mathit{received}_p \leftarrow \mathit{received}_p \setminus \{(m, -)\}$
  $\mathit{deliverable} \leftarrow \emptyset$
  for each m' in $\mathit{stamped}_p$ such that $\forall m'' \in \mathit{received}_p : TS(m') < ts_p(m'')$ **do**
    $\mathit{deliverable} \leftarrow \mathit{deliverable} \cup \{(m', TS(m'))\}$
  deliver all messages in $\mathit{deliverable}$ in increasing order of $TS(m)$
  $\mathit{stamped}_p \leftarrow \mathit{stamped}_p \setminus \mathit{deliverable}$



A.2.7 Uniform destinations agreement algorithm

This algorithm uses no group membership service. It relies on the underlying consensus and reliable broadcast algorithms to tolerate process crashes.

The following pseudocode improves on [CT96]: it uses message identifiers rather than messages wherever possible and does not use sets that grow indefinitely.

Initialization:
- $received_p \leftarrow \emptyset$ *{set of messages received by process p}*
- $unordered_p \leftarrow \emptyset$ *{set of identifiers of messages received but not yet ordered by process p}* *{each message m has a unique identifier denoted by $id(m)$}*
- $ordered_p \leftarrow \epsilon$ *{sequence of identifiers of messages ordered but not yet A-delivered by p}*
- $k \leftarrow 0$ *{serial number for consensus executions}*

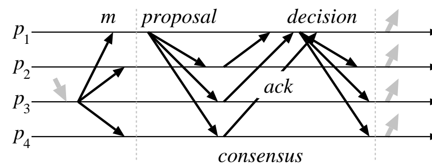
procedure *A-broadcast*(m) *{To A-broadcast a message m}*
- *R-broadcast*(m)

when *R-deliver*(m)
- $received_p \leftarrow received_p \cup \{m\}$
- **if** $id(m) \notin ordered_p$ **then**
  - $unordered_p \leftarrow unordered_p \cup \{id(m)\}$

when $unordered_p \neq \emptyset$ *{a consensus is run whenever there are unordered messages}*
- $k \leftarrow k + 1$
- *propose*$(k, unordered_p)$ *{k distinguishes independent consensus executions}*
- **wait until** *decide*$(k, idSet^k)$
- $unordered_p \leftarrow unordered_p \setminus idSet^k$
- $idSeq^k \leftarrow$ elements of $idSet^k$ in some deterministic order
- $ordered_p \leftarrow ordered_p \rhd idSeq^k$

{delivers messages ordered and received}
when $ordered \neq \emptyset$ and $\exists m \in received_p$ such that $head.ordered_p = id(m)$
- *A-deliver*(m)
- $ordered_p \leftarrow tail.ordered_p$

The figure below displays the algorithm with the Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm (see Section B.2). Reliable broadcast is depicted as a simple send to all operation.



A.3 Formulas for the contention-aware metrics

A.3.1 Non-uniform sequencer algorithm (Seq)

$$\text{Latency}_{\text{pp}}(\text{Alg. A.2.1})(n,\lambda) = 2(2\lambda+1) + (n-2)\max(1,\lambda)$$

$$\text{Latency}_{\text{br}}(\text{Alg. A.2.1})(n,\lambda) = 2(4\lambda+1)$$

$$\text{Thput}_{\text{pp}}(\text{Alg. A.2.1})(n,\lambda) = \frac{n}{(n^2-1)\max(1,\lambda)}$$

$$\text{Thput}_{\text{br}}(\text{Alg. A.2.1})(n,\lambda) = \frac{n}{(2n-1)\max(1,\lambda)}$$

A.3.2 Uniform privilege-based algorithm (PB)

$$\begin{aligned}\text{Latency}_{\text{pp}}(\text{Alg. A.2.4})(n,\lambda) =& \frac{5n}{2}(2\lambda+1)\\ &+\begin{cases}(n-2)(1-2\lambda) & \text{if } \lambda \leq \frac{1}{2}\\ \left\lfloor\frac{n-2}{2}\right\rfloor(2-2\lambda) & \text{if } \frac{1}{2} < \lambda \leq 1\\ (3-2\lambda)\max\left(0, \left\lfloor\frac{n-4}{3}\right\rfloor\right) & \text{if } 1 < \lambda \leq \frac{3}{2}\\ (\lambda-\lfloor\lambda\rfloor)\max\left(0, \left\lfloor\frac{n-4}{3}\right\rfloor\right) & \text{if } \frac{3}{2} < \lambda \leq 2\\ (\lambda-\lfloor\lambda\rfloor)\max\left(0, \left\lfloor\frac{n-3}{5}\right\rfloor\right) & \text{otherwise}\end{cases}\end{aligned}$$

$$\text{Latency}_{\text{br}}(\text{Alg. A.2.4})(n,\lambda) = \left(\frac{5n}{2}-1\right)\cdot(2\lambda+1)$$

$$\text{Thput}_{\text{pp}}(\text{Alg. A.2.4})(n,\lambda) = \frac{1}{(n-1)\max\left(1,\frac{2\lambda}{n}\right)}$$

$$\text{Thput}_{\text{br}}(\text{Alg. A.2.4})(n,\lambda) = \frac{1}{\max(1,\lambda)}$$

A.3.3 Uniform communication history algorithm (CH)

$$
\text{Latency}_{\text{pp}}(\text{Alg. A.2.5})(n,\lambda) = \begin{cases}
4\lambda + 2 & \text{if } n = 2 \\
\begin{array}{l} 6\lambda + 2 + \max(0, 2-\lambda) \\ \quad + \max(0, 1-\lambda) + \max(0, 1-2\lambda) \end{array} & \text{if } n = 3 \\
3(n-1)\lambda + 1 & \text{if } n \leq \lambda + 2 \\
\begin{array}{l} \frac{n^2-3n}{2} + 2n\lambda + \lfloor\lambda\rfloor\lambda - \frac{\lfloor\lambda+3\rfloor\cdot\lfloor\lambda\rfloor}{2} \\ \quad + \left\{ \begin{array}{ll} 1 & \text{if } \lfloor\lambda\rfloor = n-3 \\ 0 & \text{otherwise} \end{array} \right\} \end{array} & \begin{array}{c} \text{if } n \leq 2\lfloor\lambda\rfloor + 3 \\ \text{and } \lambda \geq 3 \end{array} \\
\begin{array}{l} \frac{n^2-n}{2} + 2n\lambda + \lfloor\lambda\rfloor\lambda - \frac{\lfloor\lambda+3\rfloor\cdot\lfloor\lambda\rfloor}{2} \\ \quad - \lfloor 2\lambda \rfloor - 3 + \left\{ \begin{array}{ll} \lfloor 2\lambda - 1 \rfloor & \text{if } n = 5 \\ 2 & \text{if } n = \lfloor 2\lambda + 1 \rfloor \\ 1 & \text{if } n = 7 \text{ and } \lambda = \frac{11}{4} \\ 0 & \text{otherwise} \end{array} \right\} \end{array} & \text{if } n \leq 4\lambda - 4 \\
n^2 - n + \left\{ \begin{array}{ll} 2\lambda & \text{if } \lambda < 1 \\ \left\{ \begin{array}{ll} 5\lambda - 3 & \text{if } n = 4 \\ 4\lambda - 2 & \text{if } n > 4 \end{array} \right\} & \text{if } 1 \leq \lambda < 2 \\ 5\lambda - 4 & \text{if } 2 \leq \lambda < 3 \\ 2\lfloor\lambda\rfloor\lambda - \lfloor\lambda\rfloor - \lfloor\lambda\rfloor^2 + 5 & \text{otherwise} \end{array} \right\} & \text{otherwise}
\end{cases}
$$

$$
\text{Latency}_{\text{br}}(\text{Alg. A.2.5})(n,\lambda) = 2(2\lambda+1) + (n-2)\max(1,\lambda)
$$

$$
\text{Thput}_{\text{pp}}(\text{Alg. A.2.5})(n,\lambda) = \frac{1}{(n-1)\cdot\max\left(1, \frac{2\lambda}{n}\right)}
$$

$$
\text{Thput}_{\text{br}}(\text{Alg. A.2.5})(n,\lambda) = \frac{1}{\max(1,\lambda)}
$$

A.3.4 Non-uniform destinations agreement algorithm (DA)

$$
\begin{aligned}
&\mathrm{Latency}_{\mathrm{pp}}(\text{Alg. A.2.6})(n,\lambda) \\
&= \begin{cases}
3(n-1) + 4\lambda + 2\lambda\,((n-1) \bmod 2) & \text{if } \lambda < \frac{1}{2} \\[1ex]
3(n-1) + 4\lambda + \left\{ \begin{array}{ll} 2\lambda & \text{if } n \bmod 3 = 2 \\ 2\lambda - 1 & \text{if } n \bmod 3 = 0 \\ 0 & \text{otherwise} \end{array} \right\} & \text{if } \frac{1}{2} \le \lambda < 1 \\[1ex]
\begin{array}{l} (3n-2)\lambda + 1 \\ \quad + \left\{ \begin{array}{ll} 2 + (4-n)\lambda & \text{if } n < 5 \\ 4 - 2\lambda & \text{if } n = 5 \\ X(n,\lambda) & \text{otherwise (see below)} \end{array} \right\} \end{array} & \text{if } 1 \le \lambda < \frac{3}{2} \\[1ex]
(3n-2)\lambda + 1 + \left\{ \begin{array}{ll} 2 + (4-n)\lambda & \text{if } n < 5 \\ \max(0, 4 - 2\lambda) & \text{if } n = 5 \\ 0 & \text{otherwise} \end{array} \right\} & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$
\text{where } X(n,\lambda) = \max\left(0,\ \begin{array}{l} \left\{ \begin{array}{ll} \lambda + 1 & \text{if } n \bmod 4 = 0 \\ 2 & \text{if } n \bmod 4 = 1 \\ 3 & \text{otherwise} \end{array} \right\} \\ -(n-4)(2\lambda - 2) \end{array} \right)
$$

$$
\mathrm{Latency}_{\mathrm{br}}(\text{Alg. A.2.6})(n,\lambda) = 6\lambda + 3 + (n-2)\max(1,\lambda)
$$

$$
\mathrm{Thput}_{\mathrm{pp}}(\text{Alg. A.2.6})(n,\lambda) = \frac{1}{(3n-3)\max\left(1, \frac{2\lambda}{n}\right)}
$$

$$
\mathrm{Thput}_{\mathrm{br}}(\text{Alg. A.2.6})(n,\lambda) = \frac{1}{\max\left(n+1, \frac{4n-2}{n}\lambda\right)}
$$

Appendix B

Consensus algorithms

This chapter presents the two consensus algorithms used in this thesis, due to Chandra and Toueg [CT96] and Mostéfaoui and Raynal [MR99], using the pseudocode notation of Section A.1. Both algorithms are designed for the asynchronous system model with $\diamond\mathcal{S}$ failure detectors and tolerate $f < n/2$ crashes where n is the number of participating processes. As described here, they both rely on a reliable broadcast-like primitive. For this thesis, we used the algorithm introduced next in Section B.1 for both consensus algorithms.

The Chandra-Toueg algorithm enters the reliable broadcast algorithm by calling the R-broadcast primitive. The Mostéfaoui-Raynal algorithm also calls the R-broadcast primitive but omits sending the first message of the reliable broadcast algorithm (Section 8.1.2 explains the reason). We shall denote this special type of entry as R-broadcast*.

B.1 An efficient reliable broadcast algorithm

We use an efficient algorithm inspired by [FP01]. The algorithm requires one broadcast message if the sender is not suspected (the most frequent case) and at most two broadcast messages if crashes occur but correct processes are not suspected.

The algorithm works as follows. Consider a reliable broadcast message m sent from s to $d_1, \ldots, d_k$. s simply sends m to all destinations $d_1, \ldots, d_k$. Processes buffer all received messages for later retransmission. d_i retransmits m when its failure detector suspects the sender s and all destinations with smaller indexes d_1, $\ldots$, d_{i-1}.

As for garbage collection: all processes piggyback which messages they received on outgoing messages. A message m is removed from the buffer (1) upon retransmission and (2) when its stability is detected from the piggybacked information on incoming messages.

B.2 The Chandra-Toueg $\diamond\mathcal{S}$ consensus algorithm

We applied several optimizations to the algorithm published in [CT96], in order to reduce the number of messages if neither crashes nor suspicions occur. The optimizations have been described in Section 8.1.2.

---

procedure $propose(v_p)$
  estimate$_p \leftarrow v_p$ — *{estimate$_p$ is p's estimate of the decision value}*
  $state_p \leftarrow undecided$
  $r_p \leftarrow 0$ — *{r_p is p's current round number}*
  $ts_p \leftarrow 0$ — *{ts_p is the last round in which p updated estimate$_p$ }*

  while $state_p = undecided$ **do** — *{rotate through coordinators until decision reached}*
    $r_p \leftarrow r_p + 1$
    $c_p \leftarrow (r_p \bmod n) + 1$ — *{c_p is the current coordinator}*
   Phase 1: *{all processes p send estimate$_p$ to the current coordinator}*
    if $r_p > 1$ **then**
     send $(p, r_p, estimate_p, ts_p)$ to c_p
   Phase 2: *{coordinator gathers $\lceil \frac{n+1}{2} \rceil$ estimates and proposes new estimate}*
    if $p = c_p$ **then**
     if $r_p > 1$ **then**
      wait until [for $\lceil \frac{n+1}{2} \rceil$ processes q : received $(q, r_p, estimate_q, ts_q)$ from q]
      $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
      $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
      $estimate_p \leftarrow$ select one $estimate_q \neq \perp$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
     send $(p, r_p, estimate_p)$ to all
   Phase 3: *{all processes wait for new estimate proposed by current coordinator}*
    wait until [received $(c_p, r_p, estimate_{c_p})$ from c_p **or** $c_p \in \mathcal{D}_p$] *{query failure detector $\mathcal{D}_p$}*
    if [received $(c_p, r_p, estimate_{c_p})$ from c_p] **then** — *{p received estimate$_{c_p}$ from c_p}*
     $estimate_p \leftarrow estimate_{c_p}$
     $ts_p \leftarrow r_p$
     send (p, r_p, ack) to c_p
    else — *{p suspects that c_p crashed}*
     send $(p, r_p, nack)$ to c_p
   Phase 4: *{the current coordinator waits for replies: $\lceil \frac{n+1}{2} \rceil$ acks or 1 nack. If they indicate that $\lceil \frac{n+1}{2} \rceil$ processes adopted its estimate, the coordinator R-broadcasts a decide message}*
    if $p = c_p$ **then**
     wait until [for $\lceil \frac{n+1}{2} \rceil$ processes q : received (q, r_p, ack) **or** for 1 process q: $(q, r_p, nack)$]
     if [for $\lceil \frac{n+1}{2} \rceil$ processes q : received (q, r_p, ack)] **then**
      R-broadcast $(p, estimate_p, decide)$

{if p R-delivers a decide message, p decides accordingly}
when *R-deliver* $(q, estimate_q, decide)$
  if $state_p = undecided$ **then**
   decide(*estimate*$_q$)
   $state_p \leftarrow decided$

B.3 The Mostéfaoui-Raynal $\diamond\mathcal{S}$ consensus algorithm

We applied several optimizations to the algorithm published in [MR99], in order to reduce the number of messages if neither crashes nor suspicions occur. The optimizations have been described in Section 8.1.2.

---

procedure $propose(v_p)$
  $estimate_p \leftarrow v_p$ *{$estimate_p$ is p's estimate of the decision value}*
  $state_p \leftarrow undecided$
  $r_p \leftarrow 0$ *{r_p is p's current round number}*

  while $state_p = undecided$ **do** *{rotate through coordinators until decision reached}*
    $r_p \leftarrow r_p + 1$
    $c_p \leftarrow (r_p \bmod n) + 1$ *{c_p is the current coordinator}*
    $est_from_c_p \leftarrow \bot$ *{$est_from_c_p$ is the coordinator's estimate or invalid ($\bot$)}*
    Phase 1: *{coordinator proposes new estimate; other processes wait for this new estimate}*
      if $p = c_p$ **then**
        $est_from_c_p \leftarrow estimate_p$
      else
        wait until [received $(c_p, r_p, est_from_c_{c_p})$ from c_p **or** $c_p \in \mathcal{D}_p$]
                                        {query failure detector $\mathcal{D}_p$}
        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}*
          $est_from_c_p \leftarrow est_from_c_{c_p}$

      send $(p, r_p, est_from_c_p)$ to all
    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 R-broadcasts a decide message}*
      wait until [for $\lceil\frac{n+1}{2}\rceil$ processes q : received $(q, r_p, est_from_c_q)$]
      $rec_p \leftarrow \{(q, r_p, est_from_c_q) \mid p$ received $(q, r_p, est_from_c_q)$ from $q\}$
      if $rec_p = \{v\}$ **then**
        $estimate_p \leftarrow v$
        *R-broadcast** $(p, estimate_p, decide)$ *{R-broadcast without the initial send to all}*
      else if $rec_p = \{v, \bot\}$ **then**
        $estimate_p \leftarrow v$

                                        {if p R-delivers a decide message, p decides accordingly}
when *R-deliver* $(q, estimate_q, decide)$
  if $state_p = undecided$ **then**
    decide$(estimate_q)$
    $state_p \leftarrow decided$

List of publications

Published parts of this thesis

Chapter 4

[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.

[UDS01c] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proc. 15th Int'l Conf. on Information Networking (ICOIN-15)*, pages 503–511, Beppu City, Japan, February 2001. Best Student Paper award.

Chapter 5

[UDS00b] 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.

[UDS00a] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics: Analysis of distributed algorithms. Technical Report DSC-2000-012, École Polytechnique Fédérale de Lausanne, Switzerland, February 2000.

Chapter 7

[USS03] P. Urbán, I. Shnayderman, and A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. In *Proc. Int'l Conf. on Dependable Systems and Networks*, pages 645–654, San Francisco, CA, USA, June 2003.

Chapter 9

[UDS01a] P. Urbán, X. Défago, and A. Schiper. Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be? In *Proc. 20th IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 190–193, New Orleans, LA, USA, October 2001.

[UDS01b] P. Urbán, X. Défago, and A. Schiper. Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be? Technical Report DSC-2001-037, École Polytechnique Fédérale de Lausanne, Switzerland, August 2001.

Publications related to agreement algorithms

[DSU00] X. Défago, A. Schiper, and P. Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Technical Report DSC-2000-036, École Polytechnique Fédérale de Lausanne, Switzerland, September 2000.

[DSU03] X. Défago, A. Schiper, and P. Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, 2003. Accepted for publication.

[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 Performance and Dependability Symp.*, pages 551–560, Washington, DC, USA, June 2002.

[HUSK02] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama. Performance comparison between the Paxos and Chandra-Toueg consensus algorithms. In *Proc. Int'l Arab Conf. on Information Technology (ACIT 2002)*, pages 526–533, Doha, Qatar, December 2002.

[PSUC02c] 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), supplemental volume*, pages B–32–33, June 2002. Fast abstract.

[PSUC02a] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proc. 4th European Dependable Computing Conf. (EDCC-4)*, volume

2485 of *LNCS*, pages 44–61, Toulouse, France, October 2002. Springer-Verlag.

[PSUC02b] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. Technical Report IC/2002/010, École Polytechnique Fédérale de Lausanne, Switzerland, March 2002.

[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, May 2003.

[EUS02] R. Ekwall, P. Urbán, and A. Schiper. Robust TCP connections for fault tolerant computing. In *Proc. 9th Int'l Conf. on Parallel and Distributed Systems (ICPADS)*, Chung-li, Taiwan, December 2002.

[WU02] P. Wojciechowski and P. Urbán. Modular communication infrastructure design with quality of service. In *6th CaberNet Radicals Workshop (IST-2000-25088)*, Madeira Island, February 2002.

Other publications

[PUA+97] A. Petri, P. Urbán, J. Altmann, M. D. Cin, E. Selényi, K. Tilly, and A. Pataricza. Constraint based diagnosis algorithms for multiprocessors. *Periodica Polytechnica*, 1997. Technical University of Budapest.

[APBU96] J. Altmann, A. Pataricza, T. Bartha, and P. Urbán. Constraint based system-level diagnosis of multiprocessors. In A. Hlawiczka, J. Silva, and L. Simoncini, editors, *Dependable Computing – EDCC-2*, volume 1150 of *Lecture Notes in Computer Science*, pages 403–414. Springer-Verlag Inc., 1996.

[A+03] S. Agostinelli et al. Geant4 — a simulation toolkit. *Nuclear Instruments and Methods in Physics Research*, A(506):250–303, 2003.

[FGM+97] G. Faragó, S. Gajdos, A. Máté, P. Moskovits, I. Németh, P. Urbán, and K. Wagner. Supporting an object oriented CASE tool with relational, object relational and object oriented database management systems – a case study. In *3rd Conf. Hungarian Relational Database Management System Users*, Budapest, Hungary, 1997.

Curriculum Vitæ

I was born in Budapest (Hungary) in 1974. I attended primary and secondary school in Budapest and Munich (Germany). In 1993, I graduated from the Fazekas Mihály Gimnázium, in Budapest, and started studying computer science at the Budapest University of Technology and Economics. I graduated in 1998, with a M.Sc. degree in Computing, with distinction. From 1994 onwards, I got involved in teaching and doing research about dependable multiprocessor systems. From 1996 to 1998, I spent 16 months at the European Laboratory for Particle Physics (CERN) in Geneva, Switzerland, in a C++ development project; my main tasks were the quality assurance of about 100,000 lines of code and development in geometrical modeling for my master thesis. Since 1998, I have been working in the Distributed System Laboratory (LSR) at the Swiss Federal Institute of Technology (EPFL) in Lausanne as a research and teaching assistant and a PhD student under the guidance of Professor André Schiper.