View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Concurrency and Dynamic Protocol Update for Group Communication Middleware

THÈSE N^{O} 4244 (2009)

PRÉSENTÉE LE 23 JANVIER 2009

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE SYSTÈMES RÉPARTIS

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Olivier RÜTTI

acceptée sur proposition du jury:

Prof. B. Moret, président du jury
Prof. A. Schiper, directeur de thèse
Prof. G. Candea, rapporteur
Prof. P. Felber, rapporteur
Prof. R. van Renesse, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2009

Abstract

The last three decades have seen computers invading our society: computers are now present at work to improve productivity and at home to enlarge the scope of our hobbies and to communicate. Furthermore, computers have been involved in many critical systems such as anti-locking braking systems (ABS) in our cars, airplane control systems, space rockets, nuclear power plants, banking and trading systems, medical care systems, and so on. The importance of these systems requires a high level of trust in computer-based systems. For example, a failure in a trading system (even if it is temporary) may result in severe economical losses. Hence coping with failures is a key aspect of computer systems.

A common approach to tolerate failures is to replicate a system that provides a critical service, so that once a failure occurs on a given replica, the requests to the critical service are still executed by other replicas. This approach has the advantage of masking failures, i.e., requests to the service are continuously executed even in the presence of failures. However, replication introduces a performance cost, mainly because the execution of the service requests must be coordinated among all replicas. Furthermore, despite its apparent simplicity, replication is rather complex to implement. Replication is made easier by *group communication* which defines several abstractions that can be used by the designer of replicated systems. The group communication abstractions are implemented by *distributed protocols* that compose a *group communication middleware*.

The aim of the thesis is to study two techniques to improve the performance of group communication middleware, and thus, reduce the cost of replication. First, we study *dynamic protocol update*, which allows group communication middleware to adapt to environment changes. More particularly, dynamic protocol update consists in replacing at runtime a given protocol composing the group communication middleware with a similar but more efficient protocol. The thesis provides several solutions to dynamic protocol update. For instance, we describe two algorithms to dynamically replace *consensus* and *atomic broadcast*, two essential protocols of a group communication middleware. Second, we propose solutions to introduce concurrency within a group communication middleware in order to benefit from the advantages offered by multiprocessor (or multicore) computers.

Keywords: Fault tolerance, replication, group communication, middleware, adaptive systems, distributed algorithms, consensus, atomic broadcast, dynamic protocol update, concurrency, SAMOA.

Résumé

En trois décennies les ordinateurs ont envahi notre société : les ordinateurs sont présents sur notre place de travail pour améliorer notre productivité ainsi qu'à la maison pour élargir nos possibilités de divertissements et pour communiquer. De plus, les ordinateurs sont impliqués dans de nombreux systèmes critiques tels que l'antiblocage des freins (ABS) dans nos voitures, les systèmes de contrôle aérien, les fusées, les centrales nucléaires, les systèmes bancaires et boursiers, ou encore les systèmes de santé. L'importance de ces systèmes requiert des ordinateurs avec un haut degré de fiabilité. Par exemple, une panne (même si elle est temporaire) dans un système boursier peut provoquer d'importantes pertes économiques. Pour cette raison, le traitement adéquat des pannes est un aspect clé des systèmes impliquant des ordinateurs.

Une approche usuelle pour tolérer les pannes consiste à répliquer le système sujet aux pannes. Ainsi, lorsqu'une réplique du système est victime d'une panne, les autres répliques du système sont toujours disponibles pour éxécuter les tâches attribuées au système. Une telle approche a l'avantage de masquer les pannes, c'est-à-dire que les tâches sont éxécutées sans discontinuité même en présence de pannes. Cependant, la réplication a un coût en terme de performance, un coût principalement dû à la nécessité de coordonner l'éxécution des répliques du système. De plus, malgré sa simplicité apparente, la replication est complexe à implémenter. Cette tâche est facilitée par les *communications de groupes* qui définissent des abstractions pouvant être utilisées par le developpeur d'un système répliqué. Les abstractions pour les communications de groupes sont implémentées par des *protocoles distribués* qui composent ce qu'on appelle un *intergiciel (middleware) de communications de groupes*.

L'objectif de cette thèse est d'étudier deux techniques pour améliorer la performance des intergiciels de communications de groupes, et ainsi réduire le coût de la replication. Premièrement, nous étudions le *remplacement dynamique de protocoles* qui permet aux intergiciels de communications de groupes de s'adapter aux changements d'environnement. Plus particulièrement, le remplacement dynamique de protocoles consiste à remplacer, pendant l'éxécution, un protocole faisant parti d'un intergiciel de communications de groupes par un protocole similaire mais plus performant. Dans cette thèse, nous présentons plusieurs solutions pour implémenter le remplacement dynamique de protocoles. Par exemple, nous décrivons deux algorithmes pour remplacer les protocoles de *consensus* et de *diffusion ato-*

mique, deux protocoles essentiels des intergiciels de communications de groupes. Deuxièment, nous proposons des solutions pour introduire de la concurrence dans les intergiciels de communications de groupes afin de bénéficier des avantages offerts par les ordinateurs multi-processeurs (ou multi-cores).

Mots-clés : Tolérance aux fautes, replication, communication de groupes, intergiciels, systèmes adaptifs, algorithmes distribués, consensus, diffusion atomique, remplacement dynamique de protocoles, concurrence, SAMOA.

Acknowledgments

First and foremost, I would like to thank my thesis supervisor, Prof. André Schiper. It has been a pleasure to work under his guidance during the five years that I spent in the Distributed Systems Laboratory (LSR). I am particularly grateful for all his valuable advices on writing papers and pedagogically presenting new concepts, which will be very useful to me for my future scientific (and also non-scientific) projects. I also greatly appreciated his availability, his implication, and his support, especially during the hard times preceding the submission of the papers that we co-authored (including this thesis).

Reviewing a thesis requires a lot of time and effort. For this reason and for all their interesting comments and suggestions, I am grateful to all the members of the jury, Prof. Robbert van Renesse, Prof. Pascal Felber and Prof. George Candea, as well as the president of the jury, Prof. Bernard Moret.

My gratitude also goes to all my colleagues from LSR: Fatemeh Borran, Cendrine Favez, Nuno Santos, Martin Hutle, Arnas Kupšys, Richard Ekwall, Stefan Pleisch, David Cavin, Yoav Sasson, Zarko Milosevic and Omid Shahmirzadi. During all these years that I spent with them, I enjoyed our fruitful collaborations during teaching and research activities, and I benefited from several discussions and suggestions that allow me to improve the quality of this thesis. I am also particularly appreciative to three other members of LSR. First, I would like to thank Paweł Wojciechowski for his help and his encouragements at the beginning of this thesis, and for all his contributions to this thesis as a co-author of most of the papers that are the basis of this thesis. Second, I am grateful to Sergio Mena for his valuable comments on Chapter 8, and for all his explanations on the Fortika group communication toolkit that has been extensively used as a basis of our adaptive and concurrent group communication middleware, the practical contribution of this thesis. Finally, I would like to express my gratitude to France Faille, our secretary, for her kind help dealing with traveling issues (when I attended conferences) and any other administrative tasks. I am also grateful for her patient review of the English of this thesis.

Last, but definitely not least, I wish to thank my family and my friends, for all the good moments that we shared during these five years, which largely help me to conduct this thesis to the end. In particular, I would like to express my deep gratitude to my grandparents, to my sister and her companion, and more particularly to my parents, for their continuous encouragements, their unconditional support, and the interest they have shown in my research.

Contents

1 Introduction **1**
- 1.1 Research Context and Motivation . . . 1
- 1.2 Research Contributions . . . 4
- 1.3 Structure of the Thesis . . . 6

2 System Models and Specifications **7**
- 2.1 System Models . . . 7
  - 2.1.1 Failures . . . 8
  - 2.1.2 Synchrony . . . 9
  - 2.1.3 Groups . . . 10
- 2.2 Specification of Group Communication . . . 10
  - 2.2.1 Communication Channels . . . 11
  - 2.2.2 Local Ordering . . . 11
  - 2.2.3 Failure Detectors . . . 12
  - 2.2.4 Uniform Reliable Broadcast . . . 12
  - 2.2.5 Consensus and Atomic Commitment . . . 13
  - 2.2.6 Global Ordering . . . 14

3 Dynamic Protocol Update and Related Issues **15**
- 3.1 Definitions . . . 15
- 3.2 Dynamic Protocol Update . . . 17
- 3.3 Tools for Dynamic Protocol Update . . . 18
- 3.4 Context Adaptation . . . 20
- 3.5 State Transfer . . . 21

4 Structural Issues of Dynamic Protocol Update **23**
- 4.1 Protocol Module Interactions . . . 23
- 4.2 Integration of a DPU Manager . . . 24
  - 4.2.1 Our Solution . . . 25
  - 4.2.2 Existing Solutions . . . 27
  - 4.2.3 Advantages of our Solution . . . 29
- 4.3 Protocol Addition . . . 30
  - 4.3.1 Correctness Properties . . . 30

4.3.2 Algorithm to Add Protocols 30
4.4 Protocol Removal 32
4.4.1 Safety Properties 32
4.4.2 Approach to Remove Protocols 32
4.5 Conclusion 33

5 Algorithmic Issues of Dynamic Protocol Update 35
5.1 Model for DPU Algorithms 36
5.2 DPU Specification 39
5.3 Approach to Characterize DPU Algorithms 40
5.3.1 Service Predicates 41
5.3.2 Services and Service Predicates 42
5.3.3 First DPU Instance: A Simple Replacement Algorithm . . 43
5.4 Conclusion 46

6 Algorithms for Dynamic Protocol Update of Group Communication Protocols 47
6.1 DPU Algorithm for Consensus Protocols 47
6.2 DPU Algorithm for Local Ordering Protocols 51
6.3 DPU Algorithm for Global Ordering Protocols 58
6.4 Related Work 65
6.5 Performance Evaluation 66
6.5.1 Experimental Setup 67
6.5.2 Replacement of Consensus Protocols 68
6.5.3 Replacement of Fifo Atomic Broadcast Protocols 69
6.6 Conclusion 72

7 Service Interface: A Convenient Abstraction to Implement Modular and Updateable Distributed Protocols 73
7.1 Protocol Module Interactions 74
7.2 Event-Based Protocol Frameworks 74
7.3 Service-Interface-Based Protocol Frameworks 76
7.4 Advantages of Service-Interface-Based Protocol Frameworks . . . 78
7.4.1 Protocol Module Interactions 78
7.4.2 Protocol Module Composition 79
7.4.3 Implementation of DPU Managers 80
7.4.4 Summary 81
7.5 Implementation 81
7.6 Conclusion 84

8 Transparent Concurrency for Modular Protocol Design 85
8.1 Model 86
8.2 The Module-Order Property 88
8.3 Relaxing the Module-Order Property 92

8.4 Implementation 96
8.4.1 Optimizations 96
8.4.2 Additional Features 97
8.5 Performance Evaluation 98
8.6 Related Work 102
8.7 Conclusion 104

9 Conclusion **105**
9.1 Research Assessment 105
9.2 Open Questions and Future Research Directions 107

A Scheduler for the Relaxed Module-Order Property **119**

List of Figures

3.1 An example system. 16
3.2 Effects of DPU in stack i. 17

4.1 Service calls and responses. 24
4.2 The module composition without a replacement module (left) and with the replacement module $Repl\text{-}P_i$ (right). 25
4.3 Architecture of the group communication stack. 26
4.4 Maestro. 27
4.5 Graceful Adaptation. 28

5.1 Notations for service calls and responses. 37

6.1 Calls (left) and responses (right) when no replacement occurs. . . 52
6.2 The beginning of the replacement on the initiator stack (left) and any other stack (right). 52
6.3 The end of the replacement on the initiator stack (left) and any other stack (right). 54
6.4 Latency as a function of the time at which ABcast is issued. 69
6.5 Average latency (top) and throughput (bottom) as a function of the offered load (left) or message size (right). 70
6.6 Average latency (top) and throughput (bottom) as a function of the offered load (left) or message size (right), when no replacement occurs. 71
6.7 Average latency (top) and throughput (bottom) as a function of the offered load (left) or message size (right), during the replacement phase. 71

7.1 Example of an event-based protocol stack (stack 1). 75
7.2 Example of a service-interface-based protocol stack (stack 1). . . 77
7.3 Execution of protocol interactions with interceptors (in stack 1). . 78
7.4 Implementation of a DPU manager to replace protocol P (stack 1). 80
7.5 Implementation of a group communication stack: service-interface-based (left) *vs.* event-based (right) 82
7.6 Comparison between SAMOA and Cactus: average latency (left) and throughput (right). 83

8.1 Example of computations (in stack 1). 87
8.2 Example of computations showing the restriction of module-order. 93
8.3 Example of critical event types. 94
8.4 Implementation of our benchmark. 99
8.5 Average latency (top) and throughput (bottom) as a function of the offered load when 2 (left) or 4 (right) threads execute our stack. . . 102

List of Tables

5.1 Services and corresponding predicates. . . . 43
5.2 Characterization of Algorithm 3. . . . 45

6.1 Characterization of Algorithm 5. . . . 54
6.2 Characterization of Algorithm 6. . . . 61

7.1 Service-interface-based *vs.* event-based. . . . 81

8.1 Equivalences between service-interfaces and events. . . . 96
8.2 Amount of concurrency. . . . 100

Chapter 1

Introduction

1.1 Research Context and Motivation

The last three decades have seen computers invading our society: computers are now present at work to improve productivity and at home to enlarge the scope of our hobbies or to communicate with each other. Furthermore, computers have been involved in many critical systems such as anti-locking braking systems (ABS) in our cars, airplane control systems, space rockets, nuclear power plants, banking and trading systems, medical care systems, and so on. The importance of these systems requires a high level of trust in computer-based systems. For example, a failure in a trading system (even if it is temporary) may result in severe economical losses. Hence *dependability* is a key aspect of computer systems.

Dependability. The dependability of a system can be defined by several aspects [Lap91], e.g., *reliability* (how vulnerable to failures is a system), *availability* (how long does it take for a system to be repaired after a failure), or *maintenance* (how easy is the repair of a system): dependability defines the capability of a system to cope with failures. Basically, two approaches have been considered to achieve dependability. First, *fault avoidance* consists in forecasting, preventing and removing failures of a system during the design and the implementation processes. Unfortunately, this approach does not prevent unexpected failures. Second, *fault tolerance*, which is the approach considered in the thesis, consists in designing computer systems, so that they can continue their operations even in the presence of (unexpected) failures.

Fault-Tolerance by Replication. Fault tolerance can be achieved by replicating a system (or a subpart of the system) that provides a critical service, so that, once a failure occurs on a given replica, requests to the critical service are still executed by the other replicas. This approach has the advantage to mask the failures, i.e., the requests to the service are continuously executed even in the presence of failures. However, it introduces a performance overhead, mainly because the execution of

the service requests must be coordinated among all replicas. Moreover, the coordination is rather complex to implement.

Other approaches, that are not considered in the thesis, can be used to implement fault-tolerance: e.g., *transactions* (which prevent inconsistent states of the system after failures) or *checkpointing* (which prevents losses of meaningful information after failures). Contrary to replication, these approaches require to repair the system after a failure. During the repair phase, the system cannot execute any requests. As a result, these approaches are not sufficient for systems that must remain highly available (i.e., systems in which temporary failures may have catastrophic consequences).

Numerous techniques, which can be combined, have been proposed to implement replication. Two categories can be distinguished: *hardware-based* and *software-based* techniques. The first category focuses on replicating dedicated pieces of a computer system: e.g., replicating storage devices by using the RAID [PGK88] technology. However, these techniques usually do not mask software failures. In contrast, software-based techniques address replication at the higher level of the application, which allow to efficiently mask software failures (as it has been shown in [Gra86]). The thesis contributes to the development of group communication, a well-known software-based replication technique.

Group Communication: Powerful Abstractions to Hide the Complexity of Replication. The salient issue of replication is to coordinate the execution of the different replicas. More specifically, the state of the different replicas must be maintained consistent during the overall system lifetime. Two basic schemes have been proposed to address this issue, namely *passive replication* [BMST93, GS97] (also called primary backup) and *active replication* [Lam78, Sch93]. Both schemes are quite complex to implement. This complex task is made easier by *group communication* [Bir93] which defines a set of abstractions that can be used by the designer of a replicated system. Among these abstractions, *atomic broadcast* [HT94] and *group membership* [ST06] simplify the implementation of active replication and passive replication. The common point between these two abstractions is to allow several replicas to reach an *agreement*: on the order in which requests are executed by every replica in the case of atomic broadcast, and on the primary replica chosen to execute the requests and to diffuse their results to other replicas in the case of group membership.

The problem of reaching an agreement in a group of replicas (and more generally in a distributed system) is formally defined by the *consensus* problem [Fis83]. Consensus is a quite difficult problem; for instance, it cannot be solved deterministically in an asynchronous network with a single (potential) failure [FLP85]. Consensus has been the focus of many researches. In fact, numerous algorithms to solve consensus have been proposed (see [BO83, Lyn96, CT96, Lam98, MR99] among others), each algorithm being well suited to different environments.

Finally, note that *quorum systems* [Tho79, Gif79] – another software-based technique to facilitate the implementation of replication – has been considered, especially before the introduction of group communication. However, this technique makes replication more difficult to implement than group communication, see [ES05]. Quorum systems are not considered in the thesis.

Group Communication Middleware. A large number of group communication implementations, so called group communication middleware, have been proposed over the last fifteen years (see [Bir93, WMK94, Mal96, AMMS+95, vRBM96, Hay98, Men06] among others). They are composed of distributed protocols; each protocol usually implements a specific group communication abstraction by providing a set of well-defined primitives. Starting from a monolithic design (see [Bir93, WMK94, Mal96, AMMS+95]) in which the dependencies between protocols were hidden and not clearly defined, group communication middleware has progressed to a highly modular design (see [vRBM96, Hay98, Men06]) in which protocols are considered as black boxes, with well-specified interfaces, that interact with each other. This evolution allowed group communication middleware to benefit from several common advantages of modular programming. First, it provides a high degree of configurability: the user can choose, at the initialization time, the protocols composing the middleware according to (1) the needs of the application, and (2) the properties of the underlying environment in order to optimize performance. Second, it facilitates implementation and debugging. Finally, it simplifies implementation of adaptive group communication middleware in which protocols can be replaced at runtime to improve the performance of the middleware according to the underlying environment changes, which is one topic of the thesis. Note that modular programming has also a cost: it may prevent optimizations [RMES07].

Motivation. As previously mentioned, replicating a computer system induces a performance overhead (due to the coordination of the replicas). The current thesis explores two techniques to improve performance of (modular) group communication middleware (and thus, to reduce the overhead induced by replication):

- **Concurrency.** The recent rise of multiprocessors machines, which allow to execute several computational tasks in parallel with improving performance, has made research about concurrent programming more and more popular (for instance, see the current success of the concept of software transactional memory [ST97]). Unfortunately, concurrent programming is not an easy task. Indeed, it requires to carefully synchronize the threads in order to avoid race conditions and deadlocks.

  In the context of a group communication middleware, we believe that parallelizing the processing of different network and/or application messages may result in a performance gain. Unfortunately, despite the quite large number of tools to facilitate concurrent programming (e.g., locks, monitors and software transactional memory), none is fully satisfactory in our context. This is

mainly due to (1) the specificity of the interactions between protocols within these middleware, and (2) the complexity of these middleware. In the thesis, we design a solution to easily introduce concurrency within modular group communication middleware.

- **Adaptability.** Adaptability denotes the capability of a protocol (or of the entire middleware) to adapt its strategy to environment changes, mainly to improve performance. For instance, in [MRA$^+$05], the authors show that adapting the strategy implemented by a best-effort multicast protocol significantly improves performance of a multi-user chat application. The case of atomic broadcast protocols is another example where adaptivity is a promising feature to achieve good performance. The reason is the following. A large number of strategies to solve atomic broadcast have been proposed (see [DSU04]). However, no strategy clearly outperforms all others in all environments, e.g., there is a trade-off between sequencer-based and token-based strategies [LvRB$^+$01].

  Two solutions to implement adaptability can be considered. First, group communication protocols can be implemented so that they are able to adapt themselves to the environment changes. This solution has been, for instance, implemented by the TCP protocol which adjusts its behavior according to the network congestion. However, in addition to its complexity, this solution has an inherent weakness: it allows protocols to adopt only predefined strategies, excluding new strategies (that are more efficient). In contrast, *Dynamic Protocol Update* (noted DPU hereafter) allows protocols to be updated at runtime with completely new protocols implementing better strategies. In the thesis, we consider only DPU to implement adaptability.

1.2 Research Contributions

Modular Approach to Dynamic Protocol Update (DPU). Most of current approaches to DPU (e.g., [vRBH$^+$98, CHS01]) require an explicit interaction between (1) the protocol managing and coordinating the update, and (2) the protocol that gets replaced. This interaction necessitates to implement some part of the dynamic update within the updateable protocols, which leads to poor modularity. Furthermore, it requires to understand some implementation details of the updateable protocols. In contrast, we propose an approach that is fully modular and only requires to understand the specification (rather than the implementation) of the updateable protocols.

Predicate-Based Approach to Characterize DPU Protocols. Based on our modular approach to DPU, we propose a methodology to describe the scope of applicability of DPU protocols (i.e., protocols that implement DPU). Our methodology consists in describing each DPU protocol by a set of inference rules that are based

on predicates; each predicate describes a core property of the updateable protocol. By applying the rules to the protocol that gets replaced, we can easily determine if the protocol is correctly replaced by a given DPU protocol. Moreover, the rules can be (easily) verified, which simplifies the verification of DPU protocols. To our knowledge only a single approach has concentrated on verification of DPU protocols, namely [BKvRL01]. However, this approach is limited in the sense that it allows to verify a restricted number of DPU protocols.

DPU Protocols to replace Group Communication Protocols. Based on our approaches to implement and characterize DPU protocols, we describe several DPU protocols to replace a large scope of group communication protocols, including consensus and atomic broadcast protocols, two of the most essential protocols of group communication middleware.

Service Interface: A New Abstraction to Implement Adaptive Group Communication Middleware. Group communication middleware are usually implemented using *protocol frameworks* (e.g., Cactus [Cac01, BHSC98], Appia [App01, MPR01] and Eva [BGT$^+$01]), which are programming tools that simplify the modular implementation of the different protocols composing a group communication middleware. Most protocol frameworks are based on the notion of *events* (which is the case of all frameworks mentioned above). We show that events have several drawbacks, and introduce a new abstraction, called *service interface*, that overcomes these drawbacks. For instance, contrary to events, our new abstraction provides integrated mechanisms to implement DPU protocols.

Transparent Concurrency. We identify several correctness properties for concurrent execution within a group communication middleware. Based on these properties, we describe a runtime system to transparently manage concurrency. In other words, the complexity induced by concurrency is shielded from the programmer, which greatly facilitates his/her task.

The SAMOA Protocol Framework. In order to validate our ideas, we have implemented an experimental protocol framework, called SAMOA, which facilitates the development of concurrent and adaptive group communication middleware. SAMOA is based on service interfaces (rather than on events) and provides a runtime system for transparent concurrency. We used SAMOA to implement a concurrent and adaptive group communication middleware that is based on (1) the Fortika toolkit [MRS06, Men06] and (2) the different DPU protocols presented in the thesis.

1.3 Structure of the Thesis

Preliminaries. Chapter 2 introduces basic concepts related to group communication middleware. Chapter 3 defines the problem of dynamic protocol update, and discusses several issues related to dynamic protocol update in the general context of adaptive systems. This third chapter is also an introduction of the following chapters about dynamic protocol update (Chapters 4-7).

Dynamic Protocol Update. Chapter 4 presents our modular approach to DPU, and compares it with existing approaches. Chapter 5 describes our approach to characterize DPU protocols. Chapter 6 presents several DPU protocols dedicated to the replacement of group communication protocols.

The SAMOA Protocol Framework. Chapter 7 presents the service-interface abstraction and shows its advantages over events when implementing adaptive group communication middleware. Chapter 8 studies correctness properties of concurrent executions and describes a runtime system that provides transparent concurrency.

Conclusion. Chapter 9 summarizes the contributions of the thesis, and discusses open questions and future research directions.

Chapter 2

System Models and Specifications

The current chapter introduces the basis of group communication. First, we present several system models that have been considered in this context. Second, we present and clearly specify several abstractions that are implemented by group communication protocols. Both the system model and the specifications strongly influence the remainder of the thesis, especially algorithms for dynamic protocol update (discussed in Chapters 5 and 6). Some aspects of concurrency (discussed in Chapter 8) are also influenced by the properties implemented by the group communication protocols.

2.1 System Models

We model a distributed system as a finite set of processes $\Pi = \{p_1, ..., p_n\}$. Processes do not share memory, and thus, each process has its own memory space. Processes are interconnected by communication channels, and communicate exclusively by exchanging messages through these channels. Each message is considered to be unique, and is taken from the set $\mathcal{M}$ of all possible messages. In a replicated system, the processes usually represent different host machines (each machine being dedicated to a replica of the system) interconnected by channels that are implemented by the low-level network.

A distributed system is usually defined by the assumptions made on the processes and the communication channels. We consider here three different categories of assumptions. In each category, we briefly discuss the different choices that can be established, and their consequences on the design and the properties of group communication protocols. Furthermore, we clearly state which are the assumptions that are made in the remainder of the thesis.

2.1.1 Failures

One of the salient goals of group communication is to facilitate the design of fault-tolerant systems. Therefore, the nature of the failures of both processes and channels is a key aspect of the system model. A communication channel may fail by creating, duplicating or losing messages. Because a channel can be seen as a very simple group communication abstraction, we will present more clearly which channel failures are considered in the thesis when presenting the channel specification (see Section 2.2.1).

We now describe different process failures that have been considered in the literature, starting from the most particular to the most general failures. The degree of generality of the failures is proportional to the difficulty to cope with them.

- **Crash.** Upon crashing, a process stops its execution (and thus, ceases sending and receiving messages). In the literature, two cases have been considered. First, the *crash-stop* model assumes that once a process has crashed, it never recovers (i.e., it never restores its state and continues its execution). In contrast, the *crashed-recovery* model [ACT98] allows processes to recover after a crash.

- **Send Omission.** A process omits sending a message, but continues its execution. Such failures include crashes: a crashed process can be actually considered as a process that eventually forever omits to send messages.

- **Byzantine Failure.** This category of failures includes any arbitrary failures. For instance, a Byzantine process may crash, omit message sending, alter messages, create spurious messages, or even behave maliciously (as an adversary of the group communication protocol). Furthermore, several Byzantine processes may collaborate to maximize the disturbance of the system.

In this thesis, we consider the crash-stop model only. The crash-stop model has been the focus of many researches over the last three decades, and thus, has reached maturity. As a result, the specification of group communication for such a failure model has been widely accepted. Since dynamic protocol update strongly depends on the specifications, we naturally focus on this category of failures. However, we believe that our contributions can be extended to other categories of failures.

We say that a process is *correct* if and only if it never crashes. Otherwise, the process is *faulty*. Both these predicates hold for the whole execution (i.e., a process is considered as faulty from the beginning of the execution and not only from the time of the crash). The capability to tolerate failures of a distributed protocol (i.e., the maximum number of faulty processes for which the protocol is still correct) strongly depends on the synchrony assumptions (see next section) and on the type of failures [DLS88]. This shows the importance of such assumptions on the design of group communication protocols.

2.1.2 Synchrony

The synchrony defines the timing assumptions that are made on processes and communication channels. Similarly to failures, the synchrony of a distributed system has a major impact on the design of the group communication protocols. We now discuss two extreme models (the least and the most permissive) that have been considered.

Synchronous Model. In a synchronous system, there are known bounds on (1) the relative speed of processes and (2) message transmission delays. The knowledge of the bounds greatly simplifies the design of group communication protocols. For instance, in the case of reliable channels (see Section 2.2.1), this knowledge allows processes to easily detect when another process has crashed, which allows a lot of distributed problems to be solved in presence of failures.

Such a model is however unrealistic for many practical systems. Intuitively, the reason is the following: the variation of the transmission delays inherent to distributed systems (for instance, due to unexpected loads on the network) makes the bounds hard to define in practice.

Asynchronous Model. Contrary to the synchronous model, no bounds on process speeds and message transmission delays exist in an asynchronous system. It makes it impossible to detect crash failures: a process q cannot detect whether (1) the messages sent by a process p takes an arbitrarily long time to be delivered, or (2) process p stops sending messages due to a crash. As a result, in an asynchronous system, a lot of distributed problems cannot be deterministically solved in presence of crash failures (see [FLP85, CHTCB96] among others).

In between these two extremes, several models for synchrony have been proposed. For instance, the partially synchronous model [DDS87, DLS88] weakens the synchronous model. More specifically, this model assumes that the bounds on process speeds and message transmission delays (1) hold but are a priori unknown, or (2) are a priori known but hold only after an unknown period of time. Despite these relatively weak assumptions, the partially synchronous model is sufficient to solve interesting distributed problems in presence of failures. The *timed asynchronous model* [CF99] is another example of a synchrony model. It allows interesting distributed problems to be solved within *stable* components which are, roughly speaking, subsets of processes in which the assumptions of the partially synchronous model hold.

In the thesis, we consider the asynchronous model only. This model has the advantage to be very general in the sense that any practical system implements this model. In order to be able to solve interesting problems, we augment this model with failure detectors [CT96]. Failure detectors are described in more details in Section 2.2.3.

2.1.3 Groups

Group communication protocols provide guarantees within specific groups of processes. Usually a single group of processes is considered during the whole execution. The static/dynamic assumption defines how these groups evolve during execution.

Static Model. In a static model, the group(s) of processes is fixed upon system initialization for the overall execution. This has the following drawback. Consider a protocol that tolerates a single process crash. Once a process has crashed, the protocol is no more fault tolerant (since no further crash is tolerated).

Dynamic Model. In the dynamic model, the membership of groups can change during the execution. Typically, processes that have crashed can be removed from a group, and new processes may be added to the group, e.g., to replace the crashed processes. This has the following advantage. Consider a protocol that tolerates a single crashed process in a group. In contrast to the static model, the crashed process can be replaced by a new process, so that the protocol can tolerate a new process crash after the crashed process has been replaced. However, this feature does not allow two simultaneous crashes.

When a single group is considered during the overall execution, such a model is known as the *primary partition* model. Some authors (see [CKV01, BDM01] among others) also consider a model in which, contrary to the *primary partition model*, several disjoint groups of processes can coexist during the execution. This model is known as the *partitionable model*. However, in our opinion, the specifications in the partitionable model have not reached a sufficient level of maturity [PRS08]. Therefore this model is not considered in the context of this thesis.

In the thesis, we focus on the static group model only, mainly because, contrary to other models, the specifications with static groups has been widely accepted and reached maturity. However, we think that our work is also relevant to the primary partition model, but requires some adaptations. This hope is reinforced by the fact that the specification of group communication in the primary partition model can be expressed as a generalization of the specification in the static model [Sch06].

2.2 Specification of Group Communication

We now specify group communication, starting from the simplest to the most complex abstractions. Each abstraction is clearly specified by a set of properties. Note that for each property in Sections 2.2.4-2.2.6, we can distinguish the non-uniform version (that applies to correct processes only) from the uniform version (that applies to all processes) of the property. In the thesis, we mostly consider uniform properties, and consider a non-uniform property only when the corresponding uniform property does not make sense (e.g., because it cannot be guaranteed).

2.2.1 Communication Channels

A communication channel can be seen as a special abstraction for point-to-point communication between processes, which is defined by a subset of the following properties.

No Creation. If a process q receives a message m from process p, then p has previously sent m.

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

Fair Loss. If a correct process p sends an infinite number of messages to a correct process q, then q eventually receives an infinite number of messages.

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

A *best-effort* channel ensures the Fair Loss, No Creation, and No Duplication properties. Such a channel is also called *fair-lossy* channel in literature. In contrast, a *(quasi-)reliable* channel ensures the No Loss, No Creation, and No Duplication properties.[1] Note that fair-lossy channels are strictly weaker than reliable channels.

2.2.2 Local Ordering

Local ordering provides some guarantees on the order of message deliveries. We say that the ordering is local when messages are ordered independently on each process. We now present two different local ordering properties, and specify several abstractions based on these properties. The first local ordering property, called *fifo order*, ensures that two messages sent by a given process are received in the order in which they are sent.

Fifo Order. Consider two message m and m' that are sent by a process p, such that m is sent before m'. If process q receives both messages m and m', then q receives m before m'.

Best effort fifo order and reliable fifo order are then defined by the fifo order property plus the properties ensured respectively by best effort channels and reliable channels. These two abstractions simplify the implementation of complex protocols: e.g., atomic broadcast protocols [EMS95, CHD98].

The *causal order* property is based on the *happened before* relation [Lam78].

Causal Order. Consider two message m and m', such that m happened before m'. If process q receives both messages m and m', then q receives m before m'.

[1] In literature, reliable channels sometimes denote a channel that ensures that any message sent (by a correct or a faulty process) to a correct process p is eventually received by p. However, such channels are not realistic from a practical point of view, and are only considered to prove theoretical results. We do not consider such channels in the thesis.

Causal order is strictly stronger than fifo order. Similarly to reliable fifo order, reliable causal order is defined by the causal order property plus the properties ensured by reliable channels. Causal order can be used to compute the global consistent state [CL85] of a distributed computation.

2.2.3 Failure Detectors

Failure detectors have been introduced in [CT96]. A failure detector is a special abstraction[2] that provides to each process some information on the state (i.e., correct or faulty) of other processes. More precisely, a failure detector maintains for each process p a set $suspected_p$ of processes that p suspects to be faulty. The information provided by the failure detector (i.e., all the sets $suspected_p$) can be incorrect: for instance, some correct process can be wrongly suspected by some other processes. In addition, the information is not necessarily consistent, i.e., a process p may forever suspect a process q, while a process r never suspects q.

Formally, failure detectors are defined in terms of *completeness* and *accuracy*, which determine how precise the information is about respectively correct and faulty processes. We now specify different levels of completeness and accuracy that are relevant to the thesis. Others can be found in [CT96].

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

Strong Accuracy. 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 can define the two classes of failure detectors that are considered in the thesis. The perfect failure detector $\mathcal{P}$ is defined by strong completeness and strong accuracy. On the other hand, the $\diamond\mathcal{S}$ failure detector is defined by strong completeness and eventual weak accuracy. The perfect failure detector is strictly stronger than the $\diamond\mathcal{S}$ failure detector. Augmenting an asynchronous network with one of these two failure detectors allows us to implement complex group communication abstractions (e.g., consensus, atomic broadcast and generic broadcast). Note that our contribution in the context of dynamic protocol update (and more specifically, the DPU protocol presented in Section 5.3.3) allows us to replace any classes of failure detectors defined in [CT96] and not only the ones presented above.

2.2.4 Uniform Reliable Broadcast

Roughly speaking, reliable broadcast ensures that correct processes deliver the same set of messages. This abstraction is usually used to implement more complex abstractions, such as consensus and atomic broadcast. Reliable broadcast is

[2] Usually, we use the term of *oracle* rather than the term abstraction to qualify failure detectors.

defined by the two primitives $rbcast$ and $rdeliver$ which are called when a message is respectively sent and received. Reliable broadcast is formally defined by the following three properties [HT94]:

Validity. If a correct process p rbcasts message m, then some correct process rdelivers m.

Uniform Integrity. Every process rdelivers a message m at most once and only if m was previously rbcast by some process.

Uniform Agreement. If a process rdelivers a message m, then every correct process rdelivers m.

2.2.5 Consensus and Atomic Commitment

Consensus [Fis83] and non-blocking atomic commitment [BHG87] are close to each other (a detailed comparison can be found in [CB03]). Both problems allow processes to agree on a common value that is called the decision. However, the conditions that must satisfy the decision differ (see below).

Consensus is an important problem in the context of group communication from both a practical and a theoretical point of view [MSW03]. Consensus is also a difficult problem that cannot be solved deterministically in an asynchronous system with a single (potential) process failure [FLP85].

The consensus problem is defined by the two primitives $propose$ (executed by a process to propose a value) and $decide$ (that returns the value that has been decided). Formally, (uniform) consensus is specified by the following four properties:

Uniform Integrity. A process decides at most once.

Uniform Agreement. Two processes never decide differently.

Uniform Validity. If a process decides, the decision value has been proposed by some process.

Termination. All correct processes eventually decide.

The non-blocking atomic commitment problem has been defined in the context of database systems. The problem allows several processes to agree on committing or aborting a given transaction. This problem is defined by the same properties as consensus, except for the Uniform Validity property which is modified as follows:

Uniform Validity. The decision must satisfy the following two properties. If all processes propose $commit$ and no fault occurs, then the decision must be $commit$. If one process proposes $abort$, then the decision must be $abort$.

A problem similar to the non-blocking atomic commitment problem, namely atomic commitment, has been considered in the crash-recovery model. Atomic commitment is defined by all properties of non-blocking atomic commitment except Termination. This implies that protocols solving atomic commitment (e.g., the two phase commit protocol [BHG87]) do not necessarily terminate (in the case of

failures). In the remainder of the thesis, we only consider the non-blocking atomic commitment problem. For readability, we later refer to it as the atomic commitment problem.

2.2.6 Global Ordering

Global ordering provides guarantees on the order of message deliveries. Contrary to local ordering, the order of deliveries is (partially) common to all processes (i.e., the ordering is not performed independently on each process). We now describe two abstractions that provide global ordering, namely uniform atomic broadcast [HT94] and uniform generic broadcast [PS02]. Both are specified by the properties of uniform reliable broadcast plus one specific global ordering property.

Informally, atomic broadcast ensures that all processes deliver any pair of messages in the same order. Atomic broadcast is a powerful abstraction to implement replicated state machines [Sch93] or active replication. Formally, it is defined by the two primitives $abcast$ and $adeliver$, and the properties of uniform reliable broadcast plus the following property:

Uniform Total Order. For any two processes p and q and any two messages m and m', if p adelivers m before m', then q adelivers m' only after having adelivered m.

In the thesis we also consider a variant of uniform atomic broadcast, which is called uniform fifo atomic broadcast [HT94]. Uniform fifo atomic broadcast is defined by the properties of uniform atomic broadcast plus the fifo order property (defined in Section 2.2.2).[3]

Uniform generic broadcast can be seen as a generalization of uniform atomic broadcast and uniform reliable broadcast. Basically, uniform generic broadcast ensures the same ordering guarantees as uniform atomic broadcast, but only for *conflicting* messages. The conflict relation is defined by the application, and is non-reflexive and symmetric. Formally, uniform generic broadcast is defined by the two primitives $gbcast$ and $gdeliver$ and by the properties of uniform reliable broadcast plus the following property:

Uniform Generic Order. For any two processes p and q and any two conflicting messages m and m', if p gdelivers m before m', then q gdelivers m' only after having gdelivered m.

Note that if all messages conflict, uniform generic broadcast is strictly equivalent to uniform atomic broadcast. On the other hand, if no message conflicts, it is equivalent to uniform reliable broadcast. This explain why this abstraction can be seen as a generalization of uniform atomic broadcast and uniform reliable broadcast.

[3] Note that our specification of uniform fifo atomic broadcast slightly differs from the one in [HT94]. However these two specifications are equivalent.

Chapter 3

Dynamic Protocol Update and Related Issues

This chapter aims at introducing the problem of dynamic protocol update (DPU) which is the focus of the following chapters. We also clearly position the DPU problem with respect to other issues that have been considered in the context of adaptive group communication middleware and, more generally, adaptive distributed middleware. We distinguish three categories of issues. First, we discuss *tools for dynamic protocol update*, i.e., middleware or programming languages that facilitate the implementation of dynamic protocol update. Second, we focus on *context adaptation* that denotes the self-adaptation of group communication middleware. Finally, we briefly present the problem of *state transfer* which deals with the transfer, during DPU, of meaningful information from (1) the protocol that gets replaced to (2) the new protocol.

3.1 Definitions

We introduce below simple definitions that clearly differentiate the implementation from the specification of group communication abstractions. This allows us to clearly define DPU. Note that both the definitions and the notations that are introduced in this section hold for the remainder of the thesis. These simple definitions will be, however, extended later when necessary.

Distributed Protocols and Protocol Modules. A *distributed protocol* is an implementation of a group communication abstraction. Distributed protocols are composed of a set of identical *protocol modules*, each module running on a different process. Modules may maintain some data. Modules of the same protocol may communicate with each other (by exchanging messages across the network).

Protocol Stacks. The set of modules (of different protocols) located on a process is called a *protocol stack*. Modules can be dynamically *added* or *removed* to/from a stack. Modules in a stack may interact with each other. However, they do not

share data. In other words, accessing to the data of a given module requires *explicit* interactions with this module.

Note that, despite its name, a stack is not strictly layered, i.e., a module may interact with all other modules in the same stack, not only with the modules directly above and below. In the remainder of the thesis, we use the terms *process* and *stack* interchangeably.

Services. A *service* denotes the specification of a group communication abstraction. Thus, a protocol P can be seen as the implementation of some service p. We say that protocol P *provides* service p. Similarly, we say that a module of protocol P located on stack i provides service p on stack i. For example, the service *s-atomic-broadcast* is provided by a protocol *p-atomic-broadcast* represented by a module *m-atomic-broadcast* on each stack. A protocol P providing some service p may also *require* some other services.

Module Bindings. A module can be dynamically *bound* to a service that it provides. It can be later *unbound*. Unbinding a module does not remove it from the stack. Stacks may contain several modules that provide the same service p. However, at most one module in a stack is bound to a given service p at a time.

Module bindings define the possible interactions between protocol modules in a stack. A module requiring service p can interact with the module that is bound to p. More precise definitions of protocol module interactions will be introduced in the following chapters. Note that we may introduce different definitions for protocol module interactions, each definition being adapted to the work that is described. This allows us to simplify the presentation of our contributions, but does not influence the coherence of the thesis.

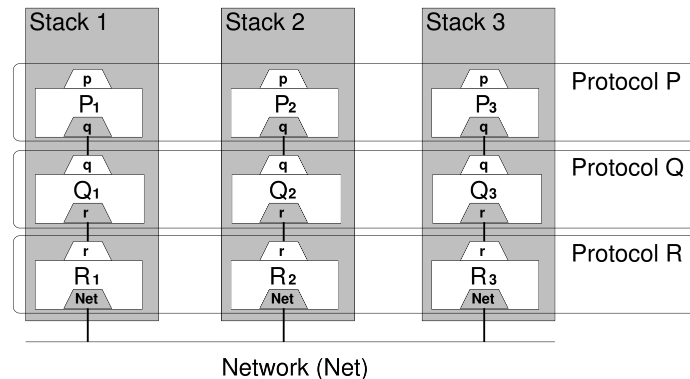

Figure 3.1: An example system.

Figure 3.1 shows an example system. Protocols are represented with capital letters P, Q, and R, and services with small letters p, q, and r. We write P_i to denote a module of the protocol P, which is part of stack i ($i = 1, 2, ..$). Modules are illustrated in figures as boxes. Services that are provided by a module are

named in white trapezoids that are aligned outside the box of the module. Similarly, a gray trapezoid named q inside the box representing a module P_i indicates that P_i requires service q. For example, module Q_1 provides service q and requires service r (see Fig. 3.1). Finally, a link between the service provided by some module Q_i and service required by some other module P_i shows that module Q_i is bound to the service required by P_i. Note that we consider the network as being a service (named *Net*).

3.2 Dynamic Protocol Update

We now define dynamic protocol update (DPU). Consider a service q. Furthermore, assume that on each stack i, a module Q_i is bound to service q (and thus, protocol Q provides q). Imagine now that we want to replace protocol Q by a new protocol $newQ$ that also provides service q. Dynamic protocol update consists in unbinding module Q_i and binding new module $newQ_i$ on each stack $i \in \Pi$ while maintaining the properties of service q. We require dynamic protocol update to be as smooth as possible. In other words, we would like to reduce as much as possible the impact of the update on the performance of the group communication stack.

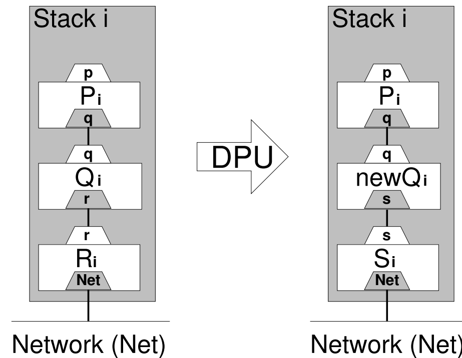

Figure 3.2: Effects of DPU in stack i.

If the new protocol $newQ$ requires a new service s that is not provided by any other protocol, a new protocol S that provides service s has to be added during the dynamic update of protocol Q. We illustrate this example in Figure 3.2. Note that to keep Figure 3.2 readable, we have removed modules Q_i and R_i from stack i after DPU. This does not mean that these modules must necessarily be removed from stack i after DPU.

DPU Dimensions. We consider two complementary dimensions of DPU: (1) the structural dimension and (2) the algorithmic dimension. These two dimensions are discussed separately in this thesis. However, it should be noted that these dimensions are not fully orthogonal, i.e., they cannot be addressed independently from each other.

- **The structural dimension** deals with the integration, into the system, of a DPU manager which performs the DPU. In addition, the structural dimension is related to the addition and the removal of protocol modules in protocol stacks. We discuss structural issues of DPU in Chapter 4.

- **The algorithmic dimension** deals with the DPU algorithm implemented by the DPU manager. Furthermore, the algorithmic dimension concerns specifications and correctness proofs of DPU algorithms. These issues are discussed in Chapter 5. Moreover, we present several examples of DPU algorithms in Chapter 6.

Discussion. One can observe that our definition of DPU is limited in the sense that we consider that the protocols on which DPU is applied (i.e., Q and $newQ$) ensure exactly the same specification (i.e., Q and $newQ$ provide the same service). This explains why some authors (see [AC03, RW04] among others) consider dynamic protocol extension (DPE). The DPE problem is similar to DPU except that the specification of $newQ$ may *extend* or *differ from* the specification of Q.

We now show that DPE is equivalent to DPU in the sense that solving DPU allows us also to solve DPE. Consider a protocol R that provides service r and a protocol S that provides service s. Furthermore, assume that R is bound to service r that is required by protocol Q (that provides service q). Consider now the DPE of protocol R by protocol S. Two cases have to be considered:

- **Service s extends r.** This case is trivial. Protocol S obviously also provides service r. As a result, the DPE of protocol R by protocol S is equivalent to the DPU of R by S.

- **Service s differs from r.** Because protocol Q requires service r, protocol Q may not be able to interact correctly with the new protocol S (which provides s). Thus, protocol Q must also be updated with a new protocol $newQ$ providing service q and requiring service s. As a result, solving DPE of protocol R by protocol S is equivalent to solving DPU of Q by $newQ$.

3.3 Tools for Dynamic Protocol Update

We now briefly discuss two categories of programming tools that facilitate the implementation of dynamic update in distributed systems: *component-based middleware* and *programming languages*. We study an additional category, namely *protocol frameworks*, in more details in Chapter 7. Contrary to component-based middleware and programming languages, protocol frameworks are specifically designed to implement (adaptive) group communication middleware, and thus, are more convenient for this purpose.

Component-Based Middleware. Component-based models, such as Enterprise JavaBeans [Sun06], COM+ [Mic01], or the CORBA Component Model [OMG04], are the context of several papers about dynamic update in distributed systems. We now present some work in this domain, and discuss its relevance in the context of DPU. Let us first briefly describe the main abstractions defined in component-based models,[1] and explain how they are related to distributed protocols:

- **Components** are computational units with well-defined interfaces, that are hosted on a given process. Different components (possibly hosted on different processes) can cooperate with each other thanks to *bindings*.

- **Bindings** are (distributed) computational units with well-defined interfaces that allow collaboration of different components. A common example of bindings is remote invocation mechanisms (e.g., Object Request Broker, RMI or RPC). In a component-based model, a distributed protocol can be considered as a binding [CBCP02].

Some papers (e.g.,[HHD98, HLA03, BNS+05]) describe component-based middleware that facilitate dynamic update of components only, excluding dynamic update of bindings. Thus, these middleware cannot be considered to implement dynamic protocol update. In contrast, several authors [Led99, KRL+00, BCRP98, CBCP02] present different middleware that allow implementation of dynamic update of bindings. However, it is not clear if these middleware provide sufficient features to allow correct implementation of DPU (especially in the case of group communication protocols which, contrary to most bindings, ensures strong consistency guarantees). More details and references about dynamic update in the context of component-based middleware can be found in a survey paper on *compositional adaptation* [MSKC04].

Programming Languages. We now discuss two of the main programming languages that allow dynamic reconfiguration of distributed applications, namely Conic [MKS89] and Argus [BD93]. These languages are based on basic abstractions called *logical nodes* in Conic and *guardians* in Argus, which both represent a piece of code located on a given process. These basic abstractions can be grouped by interconnecting them using remote method invocation mechanisms or basic communication primitives. A distributed protocols can be seen a group of basic abstractions both in Argus and Conic.

Conic and Argus provide features to replace dynamically a group of basic abstractions (and thus, a distributed protocol). However, their approach is based on a preliminary phase where all basic abstractions of the group that is replaced are put in a quiescent state. This preliminary phase requires to block all interactions issued to the replaced group, and thus may result in a significant interruption of the service availability during the dynamic protocol update.

[1] Our definitions were inspired from [CBG+08] and has been significantly simplified in order to facilitate understanding of this paragraph.

Note that several languages provide features for dynamic update of software modules on a single process, i.e., without coordination support for dynamic update in distributed environments. Examples of such languages can be found in [AV90, WKG00, MPG+00, Dug01, Hic01, SHB+05].

3.4 Context Adaptation

Context adaptation is the capability of group communication middleware (or more generally distributed middleware) to adapt themselves to the environment changes. In [DL02], the authors distinguish three steps that occur during context adaptation. First, *environment observation* consists in collecting meaningful information about the underlying environment. Secondly, *decision taking* deals with choosing the most appropriate protocols according to the underlying environment. Finally, the last step is *action*, which consists in performing the necessary DPU to install the most appropriate protocols. Different solutions can be used to address the first two steps:

- **Environment Observation.** Two different solutions have been proposed to collect environment information. In the context of component-based middleware (see [KRL+00, BBI+00, DL02]), environment observation is achieved through several components, each component being dedicated to the observation of a specific resource. The information is then centralized by a dedicated component. Contrary to this solution, the work in the context of group communication middleware proposes decentralized solutions to collect the information about the environment (see [CHS01, MRA+05]). More precisely, each protocol stack contains an *observer* module that (1) collects information on the underlying environment and (2) disseminates it to the observer modules on the other stacks.

- **Decision Taking.** Existing solutions are based on *adaptation policies* (see [BBI+00, BCHS01, DL02]). Adaptation policies are, basically, lists of logical rules based on predicates that are computed from the information that is collected during environment observation. The logical rules are then used to deduce the most appropriate protocols according to the underlying environment. In this sense, the adaptation policies determine the dynamic protocol updates to be performed. In [BCHS01], the authors advocate for adaptation policies based on fuzzy logic [Zad72], which is closer to natural language than deterministic logic. This similarity with natural language facilitates the definition of efficient adaptation policies.

Note that the DPU solutions that are presented in next chapters can be applied in conjunction with the solutions described here. In other words, our solutions can be used to implement context adaption. We do not explore further context adaptation in the remainder of the thesis.

3.5 State Transfer

In order to maintain the properties of the service that is replaced during DPU, it may be necessary to transfer some data from the protocol that gets replaced to the new protocol. For instance, consider the DPU of a protocol P by a protocol $newP$ such that both P and $newP$ provide a dynamic atomic broadcast service [Sch06]. This service is similar to the atomic broadcast defined in Section 2.2.6, but is dedicated to primary-partition model. Roughly speaking, dynamic atomic broadcast ensures that messages are delivered in a total order among a set of processes, called a view, while allowing dynamic changes of the view (initiated externally to the protocols that provide the service). Thus, DPU of protocol P by protocol $newP$ requires to transfer the current view of P to $newP$. Otherwise, protocol $newP$ will not deliver messages to the correct view, and thus, the properties of dynamic atomic broadcast will not be maintained through DPU. We now discuss both theoretical and practical studies about state transfer.

Theoretical Work. In [BKvRC01, LvRB$^+$01], the authors formally characterize the distributed protocols that do not require state transfer upon DPU. Informally, the characterization states the following. Consider a protocol P that provides a service p. Protocol P does not require state transfer if interactions with a protocol Q that requires p are not influenced by past interactions with this protocol Q.

Practical Work. In [SPW03], the authors define an approach for state transfer that requires every module to implement two methods: the first one *externalizes* the meaningful state of the module while the second one *internalizes* a given state in the module. For instance, in the case of dynamic atomic broadcast module, these methods respectively returns and sets the current view of a module. Similar solutions are described in [BBI$^+$00, KRL$^+$00, CHS01, HLA03]. In [LC05], the authors describe a method for state transfer that uses JAVA serialization [Sun04]. Unfortunately, none of these papers formally describe the properties that are ensured by the state transfer. This makes it difficult to reason about the correctness of DPU with such solutions.

In this thesis, we do not discuss the problem of state transfer.

Chapter 4

Structural Issues of Dynamic Protocol Update

This chapter focusses on the structural dimension of DPU (see Section 3.2). More specifically, we discuss here the three following issues that are all related to the changes of the stack structure induced by DPU:

- **Integration of a DPU Manager.** We show in Section 4.2 that existing solutions do not satisfactorily integrate a DPU manager in a system. For instance, in [vRBH$^+$98, CHS01], the DPU manager explicitly interacts with the updateable protocols, which leads to poor modularity. We propose a new solution and show that it has several advantages over existing solutions.
- **Protocol Addition.** Before updating a given protocol with a new protocol $newP$, the modules of $newP$ must be added in all stacks. This task is precisely what we call *protocol addition*. We define in Section 4.3 correctness properties for protocol addition in distributed systems. Based on these properties, we provide a simple algorithm for protocol addition.
- **Protocol Removal.** After having unbound the modules of the protocol P that gets replaced, modules of P may be removed from all stacks. We call this operation *protocol removal*. In Section 4.4, we introduce safety properties which a protocol must ensure in order to be removed from the system. We then discuss the implementation of protocol removal.

4.1 Protocol Module Interactions

We consider two kinds of protocol module interactions in our theoretical work about DPU that is presented in Chapters 4 to 6: *service calls* and *service responses* (see Figure 4.1). Informally, service calls allow a module to perform requests to the services it requires. On the other hand, service responses denote interactions that correspond to replies resulting from service calls.

Service calls. When a service q is called, the module Q_i that is bound to q is executed. If no module is bound to q, the service call is blocked until some module is bound to q.

Service responses. Consider a call to service q issued by some module P_i on stack i. Assume that the module that is bound to q on stack i is module Q_i (i.e., Q_i executes the call). We define the *responses* to this call to be any invocations of a module P_j by Q_j in some stack j ($j = i$ or $j \neq i$) that result from the initial call.[1] Note that a call may result in several service responses (on the same stack or/and on different stacks).

If module P_j is not currently in stack j upon a service response, then the response issued by Q_j is delayed until P_j is added to stack j. However, we assume that the module Q_j that issues the response is not blocked while the response is delayed. Finally, note that a module Q_i can respond to a service call even if Q_i is not bound to service q at the time of the response (see module Q_3 in Figure 4.1).

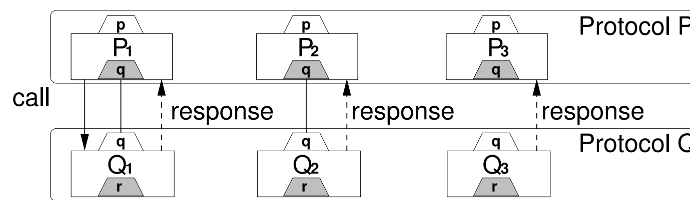

Figure 4.1: Service calls and responses.

Figure 4.1 illustrates service calls and responses. The call of a service q made by module P_1 is shown with a solid arrow. Responses to this call are represented with dashed arrows. Note that a call to service q can be seen as an interaction within a stack that occurs between the module P_1 that calls q and the module Q_1 that executes the call. On the other hand, the responses to that call represents an interaction within a protocol, e.g., interaction of P_1 with P_2 and P_3.

4.2 Integration of a DPU Manager

This section first describes our solution which we illustrate in the context of a group communication middleware. In a second step, we discuss representative existing solutions. Based on these descriptions, we finally show that our solution has several advantages over existing solutions.

[1] If service q is required by several modules in stack j, we assume that module Q_j knows the module P_i that initiates the call. This allows module Q_j to invoke the correct module P_j upon responses to the initial call. We discuss how to implement this feature in Chapter 7.

4.2.1 Our Solution

The basic idea of our solution is to add a level of indirection between the protocol P that gets replaced and the protocols that use protocol P. To do so, for each updateable protocol P, a DPU module is added to each stack in order to intercept the calls and responses to/from the service provided by P.

Note that similar solutions are presented in literature. For example, the approach described in [BKvRL01] also intercepts service calls and responses issued to/by the protocols that get replaced. However, the solution is designed for strictly layered stacks, and thus, is less general than our solution. In [MR06], the authors describe a solution to replace fifo atomic broadcast protocols which is based on an approach to integrate a DPU manager that is very similar to ours.

Description. Consider a protocol P that has to be replaced. Moreover, assume that the service p provided by P is required by two protocols Q and R. Thus, in each stack i in a system without a DPU manager (see Figure 4.2, left), modules Q_i and R_i call directly service p. Similarly, responses from service p issued by module P_i are directly executed by module Q_i or R_i.

On the other hand, in each stack i in a system that integrates our solution for DPU (see Fig. 4.2, right), the modules Q_i and R_i do not call service p directly but via another service r-p provided by our DPU module (*Repl*-P_i). Thus, modules Q_i and R_i are slightly modified to require service r-p.[2] Moreover, module *Repl*-P_i requires service p, so that it can forward the calls of service r-p to the module that is bound to p. Similarly to service calls, service responses from service p transit by the module *Repl*-P_i. Note that modules *Repl*-P_i may require some services in addition to service p.[3]

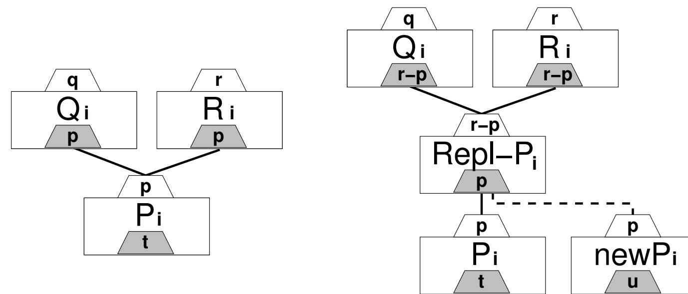

Figure 4.2: The module composition without a replacement module (left) and with the replacement module *Repl*-P_i (right).

On the right of Figure 4.2, we show the local replacement of module P_i by module $newP_i$. The dashed line connecting modules *Repl*-P_i and $newP_i$ shows that *Repl*-P_i will bind $newP_i$ to the service p after having unbound module P_i from that service. Note that modules P_i and $newP_i$ do not necessarily require the

[2]This change does not require to modify the algorithm implemented by modules Q_i and R_i.

[3]We do not show this explicitly on the figure for clarity reason.

same services. Section 4.3 discusses how module $newP_i$ can be correctly added to some stack i, even if $newP_i$ requires different services than P_i.

Characteristics of our solution. First, our solution is fully modular, since it decouples (1) the implementation of the DPU modules ($Repl$-P_i) from (2) the implementation of the updateable modules (P_i and $newP_i$). Second, the implementation of module $Repl$-P_i depends only on the service provided by the updateable modules. Indeed, module $Repl$-P_i is not aware of the algorithm implemented by updateable modules P_i and $newP_i$.

Example. Figure 4.3 shows the architecture of a group communication stack that allows both atomic broadcast and consensus protocols to be updated; it builds on the $Fortika$ toolkit [MRS06, Men06] which provides various implementations of several group communication abstractions (that are described in Section 2.2). This stack is used in order to evaluate the overhead induced by the DPU algorithms described in Chapter 6:

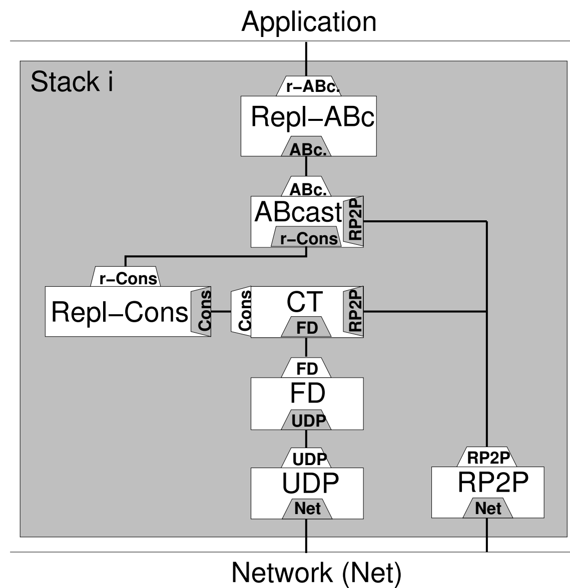

Figure 4.3: Architecture of the group communication stack.

- The *UDP* module implements best-effort channels.
- The *RP2P* module implements reliable channels.
- The *FD* module implements a *failure detector*; we assume that it ensures the properties of the $\Diamond S$ failure detector [CT96].
- The *CT* module provides a *distributed consensus* service using the Chandra-Toueg $\Diamond S$ consensus algorithm [CT96] based on a rotating coordinator.

- The *ABcast* module implements the *atomic broadcast* service.
- The *Repl-Cons* module allows replacement of protocols that provide the distributed consensus service (see Section 6.1).
- The *Repl-Abc* module implements a replacement algorithm for the atomic broadcast service (see Section 6.3).

4.2.2 Existing Solutions

Many solutions to integrate a DPU manager exist [vRBH+98, BBI+00, CHS01, SPW03, LC05, MRA+05, KG07]. However, some of these solutions (e.g. [SPW03, KG07, LC05]) are clearly not satisfactory. In [SPW03] the authors propose a solution that uses a centralized DPU manager, which limits its tolerance to failures. The approach presented in [KG07] assumes that no failure occurs during DPU. Finally, the solution described in [LC05] considers replacement in a single stack only.

We present now two example solutions to integrate a DPU manager, namely (1) Maestro [vRBH+98] implemented within the Ensemble group communication toolkit [Ens01, RBH+98], and (2) Graceful Adaptation [CHS01] implemented within the Cactus protocol framework [Cac01, BHSC98]. An approach described in [MRA+05] is similar to Maestro, but implemented within the Appia protocol framework [App01, MPR01]. In the context of component-based middleware, [BBI+00] proposes a solution that is similar to Graceful Adaptation.

Maestro [vRBH+98]. Maestro supports only the replacement of complete protocol stacks, i.e., in order to replace a single protocol, the whole stack has to be replaced.

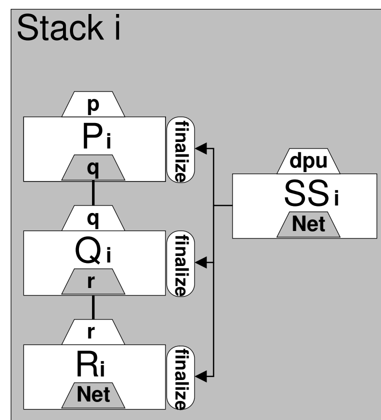

Figure 4.4: Maestro.

Figure 4.4 illustrates the solution implemented within Maestro. The main idea of this solution is to add on each stack a *stack switch module* (represented by the module SS_i in Figure 4.4). The stack switching module has the two following roles: (1) finalization of the modules of the current local stack, and (2) coordination of the start of the new modules. In order to finalize the old stack, some protocol modules must be *extended* with a method finalize that properly terminates the protocols.[4] This is represented in Figure 4.4 by rounded boxes beside protocol modules. The arrows from module SS_i to finalize boxes denote the call of module SS_i to the methods finalize which occurs each time a stack replacement is initiated.

Graceful Adaptation [CHS01]. Figure 4.5 illustrates the Graceful Adapation solution. This solution is based on special modules[5] that can be adapted (see module AQ_i). Each adaptive module AQ_i is composed of two different kinds of modules: one *module adaptor* (MA_i) and several *adaptive-aware modules* (Q_i and $newQ_i$) that provide alternative implementations of the service provided by the adaptive module. Note that modules Q_i and $newQ_i$ must require the same services (that are also required by module AQ_i).

Upon calls to the service q provided by AQ_i or responses from the services r required by AQ_i, only the adaptive-aware module (Q_i or $newQ_i$) that is *activated* is executed. Only one adaptive aware module is activated at a time. In Figure 4.5, module Q_i is activated. This is shown by links between the service provided (resp. required) by AQ_i and the service provided (resp. required) by Q_i.

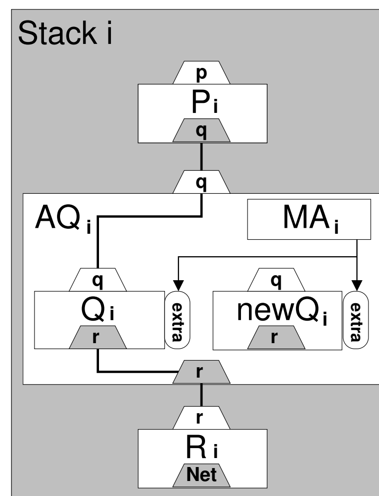

Figure 4.5: Graceful Adaptation.

[4] In [vRBH+98], the authors do not clearly define which protocols require to be extended. Note that the finalize method has several purposes, i.e., it is also dedicated to the implementation of *view changes* in the context of a dynamic group model (see Section 2.1.3).

[5] In [CHS01], the authors use the term *component* instead of *module*.

The role of the module adaptor MA_i is to perform DPU between the different adaptive-aware modules Q_i and $newQ_i$. This is done in three steps:

1. **Preparation.** The module adaptor asks the old and the new adaptive-aware modules to prepare respectively deactivation and activation. The preparation phase must be terminated on all stacks before the module adaptor starts the *deactivation* and *activation* steps. This is ensured by a *barrier synchronization* protocol. Note that barrier synchronization is executed in parallel with the execution of adaptive modules, which reduces the impact of the preparation step on service availability.

2. **Deactivation.** The module adaptor deactivates the adaptive-aware module that is currently activated (Q_i).

3. **Activation.** The old adaptive-aware module (Q_i) activates the new adaptive-aware module ($newQ_i$). The old adaptive-aware module may also transfer some state to the new adaptive-aware module.

In order to perform these three steps, the old adaptive-aware module, the new adaptive-aware module and the module adaptor communicate with each other. Thus, each adaptive-aware module must be *extended* in order to communicate with the module adaptor and some other adaptive-aware modules during DPU. This is shown in Figure 4.5 with rounded boxes beside the adaptive-aware modules Q_i and $newQ_i$.

4.2.3 Advantages of our Solution

Our solution has several advantages over existing solutions due to the way we integrate a DPU manager in protocols stacks. Our comparison is based on three perspectives, namely *modularity*, *generality* and *flexibility*:

- **Modularity.** In Maestro and Graceful Adaptation, updateable protocols must be *extended* so that the DPU manager can perform the replacement. In other words, replacement is not fully implemented by the DPU manager, which leads to poor modularity (since updateable protocols implement some part of the replacement). Our solution does not require to extend updateable protocols. As a result, the DPU modules implement entirely the replacement. Our solution is therefore modular in contrast to existing solutions.

- **Generality.** In our solution, the algorithm implemented by the DPU module depends only on the *specification* (i.e., the service) of updateable protocols. In the Maestro and Graceful Adaptation solutions, implementation of DPU relies on the *algorithm* implemented by the updateable protocols, since such protocols implement some part of the replacement. Hence, our solution is more general in the sense that a (complete) implementation of DPU allows to replace (1) all protocols ensuring a given specification instead of (2) one protocol implementing a given algorithm.

- **Flexibility.** In contrary to Graceful Adaptation, our solution does not limit the possible replacements by imposing restrictions on the services that updateable protocols may require. Unlike Maestro, replacement of a single protocol in our system does not require a whole protocol stack to be also replaced. In other words, contrary to other solutions, our approach to DPU allows flexible and fine-grained reconfiguration of protocol stacks.

4.3 Protocol Addition

We have seen in the previous section how to add to a system a DPU manager that performs the replacement of a given protocol with a new protocol $newP$. In this section, we discuss the initial phase of the replacement, which consists in adding $newP$ in the system (more precisely, adding a module $newP_i$ in each stack i). We first present basic properties that must be ensured upon addition of new protocols. Then, we describe a simple algorithm to ensure our basic properties.

4.3.1 Correctness Properties

We first introduce two properties that guarantee correct module interactions within (1) stacks and (2) protocols. First, the *stack-operationability* property ensures that no call is forever blocked. Second, the *protocol-operationability* ensures that no response for some module P_i of a given protocol P is forever delayed (except if protocol P is later removed).

Stack-operationability. A stack i is *operational* if and only if whenever a module P_i is added to i, then for each service q that is required by P_i there is a module Q_i that is bound to q.

Protocol-operationability. A protocol P is *operational* in a set of stacks Π, if and only if whenever a module P_i is added to some stack i, then a module P_j is eventually added to all correct stacks $j \in \Pi$.

Our last correctness property ensures coherence of initial bindings, i.e., the first module that is bound to a service is the same on each stack.

Initial-binding-coherence. For each service p, the first module bound to p is the same on each stack $i \in \Pi$.

Note that the specification of DPU algorithms ensures that the next bindings (not only the initial ones) are coherent (see the Replacement Order property in Section 5.2).

4.3.2 Algorithm to Add Protocols

Algorithm 1 implements the procedure $addProtocol$, which is called each time a new protocol has to be added in the system. When this procedure is called with

some protocol $newP$ as a parameter (line 1), a message ($addProt$, $newP$) is sent to all stacks using the reliable broadcast service (line 2). The tag $addProt$ indicates that protocol addition is requested. Upon delivery of message ($addProt$, $newP$) on stack j (line 3), procedure $create_module$ is executed (line 5). This procedure first checks that there is a module bound to each service q required by module $newP_j$. If this it not the case, a module Q_j is first created by calling the $create_module$ procedure, and then bound to service q (lines 9-11). We assume that for each such service q, each stack j creates and binds the same module Q_j.[6] Finally, procedure $create_module$ terminates by adding module $newP_j$ to stack j (line 12).

Algorithm 1 Protocol Addition: code of stack i.

```
 1: upon addProtocol(newP) do
 2:   RBcast(addProt, newP)

 3: upon Rdeliver(addProt, newP) do
 4:   newP_i ← local module of newP
 5:   create_module(newP_i)

 6: procedure create_module(P_i)
 7:   for all s ∈ services required by P_i do
 8:     if no module is bound to service q in stack i then
 9:       find a module Q_i providing q        {on each stack, the same module Q_i is chosen}
10:       create_module(Q_i)
11:       Q_i.bind()
12:   add P_i to stack i
```

We now prove that our algorithm ensures our correctness properties.

Initial Binding Coherence. Trivially ensured by lines 9-11. □

Stack-operationability. Obvious from lines 6-12. □

Protocol-operationability. Consider a module P_i (providing service p) that is added in stack i at line 12. Thus, stack i has called the procedure $create_module$ with module P_i as a parameter at line 5 or at line 10. In the second case, this implies that the procedure $create_module$ has been called at line 5. Let be Q_i the module passed in parameter at line 5 on stack i ($Q_i = P_i$ or $Q_i \neq P_i$). Thus, stack i has delivered a message ($addProt$, Q). Due to the properties of reliable broadcast, each correct stack j eventually executes line 5 with a module Q_j as a parameter. The two following cases have to be considered.

- $Q_j = P_j$. By lines 6-12, module P_j is added to stack j at line 12.
- $Q_j \neq P_j$. This means that on stack i, module P_i has been added by recursively calling the procedure $create_module$ at line 10. By lines 8-11, module P_i is the first module that is bound to service p in stack i. By the initial-binding-coherence-property, either module P_j has already been added in stack j or it is added by recursive calls to the $create_module$ procedure. □

[6] This can be easily ensured by defining, on each stack, the default protocol for each service.

4.4 Protocol Removal

While protocol addition is the initial step of DPU, protocol removal is the final step of DPU. Protocol removal consists in removing all modules of a given protocol $oldP_i$. This section introduces safety properties that a protocol must ensure in order to be removed. We also discuss an approach for protocol removal that ensures these properties.

4.4.1 Safety Properties

We present first two properties that guarantee that a given protocol can be removed safely. These two properties hold in any context in the sense that they define safe protocol removal independently from the DPU manager considered. The first condition, namely *call-completeness*, ensures that no more call is executed by the protocol to be removed. Note that we consider that once a protocol has been unbound from a given service, it cannot be bound anymore.[7] On the other hand, the *response-completeness* property ensures that no more response is initiated by the protocol to be removed.

Call-Completeness. Consider a protocol P. Protocol P is *call-complete* at time t if in each stack i, module P_i has been unbound from service p before time t.

Response-Completeness. Consider a protocol P. Protocol P is *response-complete* at time t if in each stack i, module P_i will not issue any response after time t.

If we consider our solution to integrate a DPU manager, the response-completeness property can be slightly weakened. Since our DPU modules intercept responses of updateable protocols, it may happen that the protocol that is replaced continues to issue responses, while these responses are discarded by the DPU modules (in other words, such responses do not influence the behavior of the DPU modules). The *response-safety* property is based on this observation.

Response-Safety. Consider a protocol P and a DPU protocol $Repl\text{-}P$ that allows to dynamically update P. Protocol P is *response safe* at time t if in each stack i, all responses issued by P_i after t are discarded by $Repl\text{-}P_i$.

4.4.2 Approach to Remove Protocols

We now present a simple approach to implement protocol removal. When a DPU module on stack i discards all responses from a module $oldP_i$ that has been previously unbound, it sends a message ($removeProt$, $oldP$) to all stacks. We show in Chapter 6 (with concrete DPU algorithms) when this message has to be sent. Upon reception of the message ($removeProt$, $oldP$) from all non-crashed stacks $j \in \Pi$, the DPU module in stack i can safely remove module $oldP_i$. Indeed, at this time,

[7] Otherwise, detecting when no more call is executed by a given protocol P requires to predict the future (and thus requires an omniscient observer).

protocol $oldP$ is both call complete and response safe. Note that this approach requires a perfect failure detector [CT96].

4.5 Conclusion

In this chapter, we have discussed several issues related to the structural dimension of DPU. First, we showed that existing solutions to integrate a DPU manager in a distributed system suffer from major drawbacks. More specifically, the most representative existing solutions (1) reduce modularity of protocol stacks, (2) result in replacement based on the algorithm rather than on the specification of the updateable protocols, and (3) limit flexibility of replacement. We provided a solution that overcomes this problems. We discussed basic correctness properties for protocol addition and removal. Finally, we presented solutions to ensure these properties.

Chapter 5

Algorithmic Issues of Dynamic Protocol Update

This chapter is devoted to the algorithmic dimension of DPU (see Section 3.2). The whole chapter is based on our approach to integrate a DPU manager (see Section 4.2.1). Our solution, contrary to others that split DPU algorithms between (1) the protocol that gets replaced and (2) the DPU manager, allows us to specify and verify DPU algorithms. The contribution described in this chapter is twofold:

- **DPU Specification.** We introduce correctness properties that each DPU algorithm must ensure. Contrary to [BKvRC01] where the authors consider that a DPU algorithm is correct if and only if it is *transparent* (i.e., the properties of the service that gets replaced are maintained during DPU), we require correct DPU algorithms to be additionally *consistent* (the effects of DPU are the same on each stack) and *live* (DPU eventually terminates).

- **Approach to Characterize DPU Algorithms.** We propose a methodology to describe the characteristics of DPU algorithms (i.e., which protocols they correctly replace). The methodology consists in characterizing DPU algorithms by a set of inference rules that are easy to prove. The rules consist of predicates that apply to group communication protocols. By applying the set of rules to a given protocol P, we can simply determine if the protocol P is correctly replaced by the DPU algorithm that corresponds to these rules. Note that our methodology can be applied to several existing DPU algorithms, such as those described in [MR06, BKvRC01].

  A similar approach has been presented in [BKvRC01]. Contrary to this approach, our methodology allows us to characterize and verify DPU algorithms for reliable protocols (e.g., reliable broadcast, atomic broadcast or reliable channels). For instance, in Section 5.3.3 we characterize and verify a simple DPU algorithm that correctly replaces reliable broadcast algorithms. More complex DPU algorithms are characterized using our methodology in Chapter 6.

5.1 Model for DPU Algorithms

This section first describes service calls and responses from an algorithmic point of view. This allows us to define the class of DPU algorithms considered in the thesis. In the following sections, we discuss our approach to characterize DPU algorithms based on these definitions.

Service calls. Upon a service call, an identifier k is transmitted to the module that executes the service call. Two calls that occur on the same stack cannot have the same identifier. However, two calls that occur on different stacks can have the same identifier. For instance, consider the atomic commitment service [BHG87] which allows a set of processes to agree on committing or aborting a transaction. Each process calls the atomic commitment service to propose to commit or to abort. Thus, each process must perform the call with the same identifier in order to clearly identify that they try to commit or abort the same transaction.

We also assume that a set of stacks denoted by *dst* is given as parameter upon service calls. A call results in service responses only on stacks $i \in dst$. Note that a stack $i \in dst$ does not necessarily generate a response. For instance, in the case of a best-effort fifo order service, it is possible that some destination stack does not generate a response. Two calls with the same identifier k have also the same set of destinations dst.

Finally, additional parameters can be passed as an argument of a service call. These additional parameters may only influence the additional parameters of the corresponding responses. We denote by $C_{s,i}(k, dst, par)$ the call to service s that occurs on stack i with identifier k, set of stacks dst and additional parameters par.

Service responses. Similarly to service calls, an identifier is transmitted upon each service response. This identifier corresponds to the identifier transmitted upon the corresponding service call. In addition to an identifier, a set of stacks src is passed as an argument upon each service response. The set src denotes all the stacks on which a corresponding call (i.e., a call with the same identifier) occurs. Usually, the set src contains only one stack. However, for some services such as atomic commitment (see above) the set src may contain several stacks.

Again, similarly to service calls, additional parameters, denoted by par can be passed as an argument upon a service response. Contrary to the identifier, the parameters par passed on a service response are not necessarily identical to the additional parameters passed on the corresponding call(s). The parameters par of responses depend only on the corresponding call(s). Furthermore, the parameters par may differ among the responses with identifier k.

We denote by $R_{s,i}(k, src, par)$ the *first* response from service s that occurs on stack i with identifier k, set of stacks src, and additional parameters par. We do not introduce a notation for further responses with identifier k on stack i, since it is useless for the purpose of the thesis.

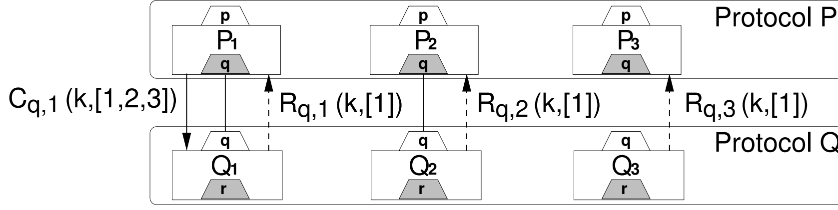

Figure 5.1: Notations for service calls and responses.

Figure 5.1 illustrates our notation for service calls and responses. The scenario consists in a call to service q made by module P_1 (shown with a solid arrow). The call is identified by k and its destination is the set of stacks [1,2,3]. Responses to this call ($R_{q,1}(k,[1])$, $R_{q,2}(k,[1])$ and $R_{q,3}(k,[1])$) are represented with dashed arrows. For clarity, we do not consider additional parameters par in this scenario.

Generic DPU Algorithm. Algorithm 2 defines the generic behavior of most algorithms presented in this thesis. All algorithms that are instances of Algorithm 2 inherit from the properties of our generic algorithm. Specifically, the DPU instances do not arbitrarily create service responses (see discussion below). As a result, if a service p (provided by the updateable protocol) guarantees that for each service response, a corresponding service call has occurred, then service r-p (provided by the DPU manager) also ensures this guarantee.

Each DPU module maintains two different sets $calls$ and $responses$: Each set contains respectively service calls and service responses. Upon a call to service r-p, the DPU module adds the call to the set $calls$ (line 8). Similarly, upon a response from p, the DPU module adds the response to the set $responses$ (line 11). Furthermore, each DPU module reacts to four categories of events: (1) calls to primitive $replaceProtocol$ for initiation of a replacement (line 5), (2) calls to service r-p (line 7), (3) responses from service p (line 10), and (4) responses from any service it requires (line 13). Upon any of these events[1], a DPU module may perform any sequence of the following basic actions:

1. Issue a call to service p (lines 16-18). The parameters of that call are taken from the set $calls$. The call is made using a new identifier ($FORWARD$, k, inf) where $FORWARD$ is a tag ensuring that the responses corresponding to the call will be added to the set $responses$ (see lines 10-11). The parameter inf, which is optional, is any other information needed for DPU.

2. Issue a response from service r-p (lines 19-21). Similarly to call to service p, the parameters of the response are taken from the set $responses$. The information inf and the tag $FORWARD$ added by a DPU module are removed from the identifier upon the response to service r-p.

[1]Each **upon** block is executed in mutual exclusion.

3. Call some service x related to the replacement task (lines 22-23). Similarly to the call of the first basic action, this call is made using a special identifier $(INTERNAL,\ k,\ inf)$ where $INTERNAL$ is a tag ensuring that the responses corresponding to the service call are not added to the set $responses$ (see lines 13-14). Again, inf is optional and denotes any other information needed for DPU. Contrary to the first basic actions, all the parameters of a call (i.e., k, inf, dst and par) are freely determined by the DPU module. Finally, note that the service x can be the service provided by the protocol that gets replaced (i.e., service p).

4. Perform locally the protocol update (lines 24-28). The local module P_i of the current protocol is first unbound from service p. Then, a module $newP_i$ of the new protocol is bound to service p and set as the current protocol module. For simplicity, we assume that module $newP_i$ is in each stack $i \in \Pi$ at the time of binding. In other words, the DPU algorithms that are considered in the remainder of the thesis do not explicitly implement protocol addition (see Section 4.3). Protocol addition can be, however, easily implemented within these DPU algorithms.

Algorithm 2 Generic DPU Algorithm: code of stack i.

```
1:  Initialisation:
2:     calls ← ∅                                            {The set of calls to service r-p}
3:     responses ← ∅                                        {The set of responses from service p}
4:     currentP_i ← initial_protocol_module                 {The module currently bound to service p}

5:  upon replaceProtocol(newP) do
6:     execute any sequence of basic actions

7:  upon r-p.call(k, dst, par) do
8:     calls ← calls ∪ (k, dst, par)
9:     execute any sequence of basic actions

10: upon p.response((FORWARD, k, inf), src, par) do
11:    responses ← responses ∪ ((k, inf), src, par)
12:    execute any sequence of basic actions

13: upon x.response((INTERNAL, k, inf), src, par) do      {x can be any service}
14:    execute any sequence of basic actions

15: List of basic actions:
16:    1) Issue a call to service p:
17:       select a call (k, dst, par) from calls according to some criterion
18:       p.call((FORWARD, k, inf), dst, par)
19:    2) Issue a response from service r-p:
20:       select a response ((k, inf), src, par) from responses according to some criterion
21:       r-p.response(k, src, par)
22:    3) Call some service x:
23:       x.call((INTERNAL, k, inf), dst, par)
24:    4) Perform locally the protocol update:
25:       currentP_i.unbind()
26:       newP_i ← local module of the new protocol
27:       newP_i.bind()
28:       currentP_i ← newP_i
```

Discussion. From lines 10-23 of Algorithm 2, one can observe that our DPU algorithms do not arbitrarily create responses. More precisely, for each response from service r-p, a corresponding call to r-p has been issued.

We now explain why we ignore the additional parameters par except if stated otherwise. Let us first define the notion of *response-coherence*. A DPU module is *response-coherent* if and only if for any identifier k, all DPU modules issue responses from service r-p (see line 21) based on responses from service p (see set $responses$ at line 20) that were issued by a module of the same protocol. It can be easily seen that all DPU algorithms in the remainder of the thesis are response-coherent for the service that they correctly replace. Furthermore, one can observe that DPU modules do not change the parameters par transmitted by service calls (see lines 7-8 and 16-18) and service responses (see lines 10-11 and 19-21). Thus, properties related to these parameters are automatically preserved by DPU modules that are response-coherent. In other words, any properties ensured by the service p that apply to the additional parameters par of both service calls and responses, are also ensured by service r-p of the DPU algorithms that are instances of our generic algorithm.

5.2 DPU Specification

We now specify the properties that each DPU algorithm must ensure. These properties ensure transparent, consistent, and terminating dynamic update for a given service p. The first property, namely Service Preservation, guarantees transparency: The service r-p provided by the DPU protocol corresponds to the service p provided by the protocol that gets replaced.

Service Preservation. Service r-p ensures the same predicates[2] as service p.

Additionally to this property, each DPU instance must ensure the following three properties. In the specification of these properties, we assume the replacement of protocol P by protocols $newP$ and $newP'$. Furthermore, for simplicity, once a module has been unbound from service p, we assume that it cannot be bound again to p.

Replacement Termination. If the primitive *replaceProtocol* is called with protocol $newP$ as a parameter on some correct stack $i \in \Pi$, then module $newP_i$ will be eventually bound to service p in stack i.

Replacement Agreement. If module $newP_i$ is bound to service p in some stack i, then in each correct stack $j \in \Pi$, a module $newP_j$ will be eventually bound to service p.

Replacement Order. If module $newP_i$ is bound to service p before module $newP'_i$ in some stack i, then each stack $j \in \Pi$ binds module $newP'_j$ to service p only after it has bound module $newP_j$ to service p.

[2]Predicates can be understood as service properties, see Section 5.3.1

The Replacement Termination property ensures *live* replacements in the sense that each DPU instance eventually finishes. Roughly speaking, Replacement Agreement guarantees that each dynamic protocol update is performed by either all correct stacks or by none of them. Replacement Order ensures that all stacks perform the local update in the same order. These two properties ensure consistent DPU, which means that if no replacement occurs anymore, the last module bound to a given service is eventually the same for each stack. This is a desirable property to allow efficient protocol removal in the sense that for each service p, it ensures that eventually all protocols providing p except one can be removed. Note that if protocol removal is not considered, Replacement Agreement and Replacement Order can simply be ignored.

5.3 Approach to Characterize DPU Algorithms

We have previously defined the generic behavior of DPU algorithms (see Algorithm 2). Several DPU algorithms for group communication protocols can be instantiated out of Algorithm 2. In this section, we introduce an approach to characterize these DPU algorithms that facilitates correctness proofs. More precisely, we show for a given DPU algorithm how to define, as exhaustively as possible, the services it correctly replaces. A trivial solution consists in simply enlisting these services and prove for each service that it is correctly replaced. Our solution is, however, more elegant, since it allows concise description of DPU algorithms and facilitates their proofs.

Our solution consists in a set of logical rules based on core properties of distributed services that we call *service predicates*. Each property provided by the services considered in this thesis – except properties related to the additional parameters that are not considered here (see Section 5.1) – corresponds to a service predicate. Our logical rules mainly allow us to determine if a DPU instance ensures Service Preservation for a given service p. More precisely, a logical rule defines the conditions on service p that ensure a predicate on service r-p, e.g. :

$$pred_1(p) \quad \wedge \quad ... \quad \wedge \quad pred_l(p) \quad \Rightarrow \quad pred(r\text{-}p)$$

Furthermore, the logical rules also describe for which services the Replacement Agreement, Replacement Order and Replacement Termination properties hold. For most DPU algorithms, these properties are ensured for any service p. In this case, the left part of the corresponding rule is equal to *true* (see Section 5.3.3).

This section is structured as follow. First, we present a (non-exhaustive) list of service predicates that we illustrate with group communication services. Then, we describe a simple example of DPU algorithm. This DPU algorithm is characterized with logical rules based on service predicates. We also illustrate the use of the logical rules with concrete examples. Finally, we illustrate correctness proofs with our example of DPU algorithm.

5.3.1 Service Predicates

Our first service predicate, namely *At-Most-One-Response*, ensures that at most one service response – corresponding to a specific call – occurs on each stack. This service predicate applies to several distributed protocols, especially to protocols that prevent multiple deliveries of the same message on a stack.

At-Most-One-Response. Consider a service s and a call $C_{s,i}(k, dst)$ that occurs on stack i. The service s satisfies the predicate *at-most-one-response* if and only if for each stack $j \in dst$, at most one response $R_{s,j}(k, src)$ occurs.

The following service predicates provide guarantees about the existence of responses related to a given service call. We define three levels for these guarantees on response reliability. The two first levels, namely *Weak Reliability* and *Strong Reliability*, are well-known properties of distributed protocols. Note that *Weak Reliability* is strictly weaker than *Strong Reliability*.

Weak Reliability. Consider a service s and an infinite number of service calls $C_{s,i}(k_1, dst)$, $C_{s,i}(k_2, dst)$, ... that occur on a correct stack i. Service s is *weakly reliable* if and only if an infinite number of corresponding responses $R_{s,j}(k'_1, src)$, $R_{s,j}(k'_2, src)$, ... eventually occur on each correct stack $j \in dst$.

Strong Reliability. Consider a service s and a service call $C_{s,i}(k, dst)$ that occurs on a correct stack i. Service s is *strongly reliable* if and only if a response $R_{s,j}(k, src)$ eventually occurs on each correct stack $j \in dst$.

Our last level for reliability, called *Group Reliability*, applies to services requiring a call from each process in the system in order to ensure a response.

Group Reliability. Consider a service s and a service call $C_{s,i}(k, dst)$ that occurs on each correct stack $i \in \Pi$. Service s is *group reliable* if and only if a response $R_{s,j}(k, src)$ eventually occurs on each correct stack $j \in dst$.

Note that a fourth level, namely *Local Reliability*, can be considered. Local reliability ensures that, for each call occurring on some correct stack i, a corresponding response occurs on the same stack i. Since this predicate is only relevant for Failure Detectors and is easily preserved by DPU algorithms, we do not consider it in this thesis.

We now present a service predicate that guarantees atomicity of responses. Informally, it ensures that a call results in either (1) no response or (2) a response on each stack in the destination set.

Response Atomicity. Consider a service s, a service call $C_{s,i}(k, dst)$ and a corresponding service response $R_{s,j}(k, src)$ that occurs on some stack j. Service s is *response atomic* if and only if a response $R_{s,l}(k, src)$ eventually occurs on each correct stack $l \in dst$.

The following four service predicates characterize ordering services. The first two service predicates guarantee that the order of service responses on a given stack

corresponds to some precedence relation defined on the corresponding calls. These two predicates only apply to services for which there is a single call per identifier.

The *Fifo Order* predicate relies on the *local precedence* relation that we now define. A call k *locally precedes* a call k' if and only if (1) both calls k and k' occur on the same stack i, and (2) call k' occurs after call k on stack i.

Fifo Order. Consider a service s, two calls $C_{s,i}(k, dst)$ and $C_{s,i}(k', dst')$, such that call $C_{s,i}(k, dst)$ locally precedes call $C_{s,i}(k', dst')$. Assume that two responses $R_{s,j}(k, src)$ and $R_{s,j}(k', src)$ occur on some stack $j \in dst \cap dst'$. Service s is *fifo ordered* if and only if response $R_{s,j}(k, src)$ occurs before response $R_{s,j}(k', src)$ on stack j.

The second ordering predicate, called *Causal Order*, is based on the causal precedence relation defined as follows. A call k *directly causally precedes* a call k' if and only if a call or a response k occurs before the call k' on stack i. The *causal precedence* relation is then defined as the transitive closure of the direct causal precedence relation. Because the causal precedence relation is strictly stronger than the local precedence relation, *Causal Order* is strictly stronger than *Fifo Order*.

Causal Order. Consider a service s, two calls $C_{s,i}(k, dst)$ and $C_{s,j}(k', dst')$ (possibly $i \neq j$), such that call $C_{s,i}(k, dst)$ causally precedes call $C_{s,j}(k', dst')$. Assume that two responses $R_{s,l}(k, src)$ and $R_{s,l}(k', src)$ occur on some stack $l \in dst \cap dst'$. Service s is *causally ordered* if and only if response $R_{s,l}(k, src)$ occurs before response $R_{s,l}(k', src)$ on stack l.

Finally, the last two service predicates, namely *total order* and *generic order*, ensure that respectively (1) all, or (2) a subset (only those that conflict), of the service responses are issued in the same order on each stack. *Total order* is strictly stronger than *generic order*. Note that *generic order* depends on a (symmetric and non-reflexive) conflict relation on service calls/responses.

Total Order. Consider a service s. Assume that a response $R_{s,i}(k, src)$ occurs before a response $R_{s,i}(k', src')$ on stack i. Service s is *totally ordered* if and only if for any stack j on which response $R_{s,j}(k', src')$ occurs, then response $R_{s,j}(k, src)$ has occured before.

Generic Order. Consider a service s. Assume that a response $R_{s,i}(k, src)$ occurs before a response $R_{s,i}(k', src')$ on stack i, such that the responses conflict. Service s is *generically ordered* if and only if for any stack j on which response $R_{s,j}(k', src')$ occurs, then response $R_{s,j}(k, src)$ has occured before.

5.3.2 Services and Service Predicates

Table 5.1 shows a list of services (that are defined in Section 2.2) in relation with the service predicates presented above. For instance, Reliable Broadcast ensures the four following predicates: At-Most-One-Response, Weak Reliability, Strong Reliability and Response Atomicity.

	At-Most-One-Resp.	Weak Reliability	Strong Reliability	Group Reliability	Response Atomicity	Fifo Order	Causal Order	Total Order	Generic Order
Best Effort Channel	+	+	-	-	-	-	-	-	-
Reliable Channel	+	+	+	-	-	-	-	-	-
Best Effort Fifo Order	+	+	-	-	-	+	-	-	-
Reliable Fifo Order	+	+	+	-	-	+	-	-	-
Reliable Causal Order [BJ87]	+	+	+	-	-	+	+	-	-
Failure Detector [CT96]	+	-	-	-	-	-	-	-	-
HO Predicate Layer [HS07]	+	-	-	+	-	-	-	-	-
Consensus [CT96]	+	-	-	+	+	-	-	-	-
Atomic Commitment [BHG87]	+	-	-	+	+	-	-	-	-
Reliable Broadcast [CT96]	+	+	+	-	+	-	-	-	-
Atomic Broadcast [HT94]	+	+	+	-	+	-	-	+	+
Fifo Atomic Broadcast [HT94]	+	+	+	-	+	+	-	+	+
Generic Broadcast [PS02]	+	+	+	-	+	-	-	-	+

Table 5.1: Services and corresponding predicates.

One can observe that Consensus and Atomic Commitment services ensure exactly the same service predicates (see Table 5.1). Indeed, the difference between the two services is related to the additional parameters – transmitted upon service calls or responses – that are not considered here (see Section 5.1). Thus, any instance of our generic DPU algorithm that correctly replaces Consensus protocols, also correctly replaces Atomic Commitment protocols, and vice versa.

5.3.3 First DPU Instance: A Simple Replacement Algorithm

We now describe a very simple DPU algorithm. We then characterize this DPU algorithm with logical rules and explain the concrete meaning of these rules. This will show that despite its (apparent) simplicity, our algorithm allows to replace several distributed protocols. Note that for simplicity, we omit protocol removal (see Section 4.4.2) from the description of our algorithm. Note that it can be rather easily modified to implement protocol removal. However, these modifications may differ depending on the service that gets replaced.

Algorithm. The basic idea of Algorithm 3 is to directly forward each call to service $r\text{-}p$ to the protocol bound to service p (see lines 7-9). A similar behavior occurs on a response from service p (see lines 10-12). Note that the sets $calls$ and

Algorithm 3 Simple DPU Algorithm: code of stack i.

```
1:  Initialisation:
2:     calls ← ∅                                          {The set of calls to service r-p}
3:     responses ← ∅                                      {The set of responses from service p}
4:     currentP_i ← initial_protocol_module               {The module currently bind to service p}

5:  upon replaceProtocol(newP) do
6:     ABcast.call((INTERNAL, newP), Π)                   {Basic action: Call a service required}

7:  upon r-p.call(k, dst) do
8:     calls ← calls ∪ (k, dst)
9:     p.call((FORWARD, k), dst)                          {Basic action: Issue a call to service p}

10: upon p.response((FORWARD, k), src) do
11:    responses ← responses ∪ (k, src)
12:    r-p.response(k, src)                               {Basic action: Issue a response from service r-p}

13: upon ABcast.response((INTERNAL, newP), j) do
14:    currentP_i.unbind()                                {Basic action: Perform locally the protocol update}
15:    newP_i ← local module of protocol newP
16:    newP_i.bind()
17:    currentP_i ← newP_i
```

$responses$ can be ignored in this algorithm. We let them appear here in order to make the link with the generic algorithm more explicit.

When a replacement is initiated on stack i (line 5), the new protocol is sent using atomic broadcast (ABcast) to all other stacks in Π (line 6). When a process executes the response of ABcast with the new protocol $newP$ (line 13), it simply (1) unbinds the current protocol module (line 14), (2) binds the new protocol module (line 16), and (3) sets the new protocol module as the current protocol module (line 17).

Characterization. Because we use atomic broadcast to diffuse the new protocol, Algorithm 3 ensures Replacement Agreement, Replacement Order and Replacement Termination for whatever service p that gets replaced (see Rules A, B and C in Table 5.2).

Concerning Service Preservation, Rules 1-4 in Table 5.2 show which predicates are ensured by service r-p when a protocol providing a given service p is replaced. For instance, if service p is weakly reliable then service r-p is also weakly reliable (see Rule 2 in Table 5.2). Thus, by the definition of Service Preservation (see Section 5.2) and Best Effort Channel services (see Table 5.1), our DPU algorithm ensures Service Preservation for the Best Effort Channel service. Because the other correctness properties of DPU are ensured independently from the service that gets replaced, best effort channel protocols are correctly replaced by our simple algorithm.

Let us now illustrate properties stated by Table 5.2 with two other protocols. First, assume that we want to replace reliable broadcast protocols. From Table 5.1, we know that the following predicates apply to Reliable Broadcast: At-Most-One-Response, Weak Reliability, Strong Reliability and Response Atomicity. From

Rule A:	*true*	⇒	Replacement Agreement
Rule B:	*true*	⇒	Replacement Order
Rule C:	*true*	⇒	Replacement Termination
Rule 1:	At-Most-One-Response(p)	⇒	At-Most-One-Response(r-p)
Rule 2:	Weak Reliability(p)	⇒	Weak Reliability(r-p)
Rule 3:	Strong Reliability(p)	⇒	Strong Reliability(r-p)
Rule 4:	Response Atomicity(p)	⇒	Response Atomicity(r-p)

Table 5.2: Characterization of Algorithm 3.

Table 5.2, all these predicates also apply to service r-p. Thus, Algorithm 3 ensures Service Preservation when replacing reliable broadcast protocols.

Imagine now that we want to replace atomic broadcast protocols. Atomic broadcast is characterized by the same predicates as Reliable Broadcast plus the Total Order predicate. Since no logical rule allows to state that Total Order applies on service r-p, we cannot conclude that atomic broadcast protocols is correctly replaced by our simple DPU algorithm.

Proofs. We prove that Algorithm 3 corresponds to the characterization in Table 5.2. The proofs related to Rules 1-4 are obvious and, thus, are omitted here. We only prove below Rules A, B, and C, i.e., the Replacement Agreement, Replacement Order and Replacement Termination properties hold for any service p (provided by the protocol that gets replaced). Note that other examples of characterization proofs are given in Chapter 6 with more complex algorithms.

Rule A: $true$ ⇒ **Replacement Agreement:** Assume that module $newP_i$ is bound to service p in some stack i. Thus, stack i executes an ABcast response with $newP$ as a parameter (see lines 13-16). From Response Atomicity of ABcast (see Table 5.1), each correct stack $j \in \Pi$ eventually executes an ABcast response with $newP$ as a parameter (line 13), unbinds the old protocol module (line 14) and finally binds the module $newP_i$ (line 15). □

Rule B: $true$ ⇒ **Replacement Order:** Assume that stack i binds the protocol module $newP_i$ to service p before binding the module $newP'_i$ (line 15). Thus, stack i executes an ABcast response R with $newP$ as a parameter before executing a similar response R' with $newP'$ as a parameter (line 13). From Total Order of ABcast (see Table 5.1), each stack j that executes response R' has executed response R before. Thus, each stack that binds the protocol module $newP'_j$ has bound module $newP_j$ before. □

Rule C: $true$ ⇒ **Replacement Termination:** Assume that $replaceProtocol(newP)$ occurs on some correct stack i. Thus, stack i executes the call to ABcast (($INTERNAL$, $newP$), Π) (line 6). From Strong Reliability of ABcast (see Table 5.1), stack i eventually executes an ABcast response with (($INTERNAL$, $newP$), i), unbinds the old protocol module and finally binds the module $newP_i$. □

5.4 Conclusion

In this chapter, we have specified the properties that DPU algorithms must ensure. Furthermore, we have proposed a methodology for characterizing DPU algorithms by logical rules, and we have shown how the approach facilitates proofs. Finally, we have provided a very simple DPU algorithm that allow us to replace some distributed protocols.

Chapter 6

Algorithms for Dynamic Protocol Update of Group Communication Protocols

In previous chapters, we proposed a general solution to (1) integrate a DPU manager in a set of stacks, and (2) characterize and verify the DPU algorithms implemented by DPU managers. The present chapter describes several DPU algorithms based on our general approach. These DPU algorithms allow us to correctly replace a large scope of group communication protocols, including consensus and atomic broadcast protocols that are the main building blocks of our group communication middleware described in Figure 4.3 (Section 4.2.1).

Our first DPU algorithm is dedicated to consensus protocols. We then describe a DPU algorithm that uses the guarantees provided by the Fifo Order predicate. This algorithm correctly replaces local ordering protocols such as reliable fifo order and reliable causal order protocols. In contrast, our third DPU algorithm is dedicated to global ordering protocols, which include generic broadcast and atomic broadcast protocols. We next discuss existing DPU algorithms. Finally, we evaluate the overhead induced by the DPU algorithms described in this chapter. The overhead is evaluated in two cases: (1) when no replacement occurs and (2) during the replacement.

6.1 DPU Algorithm for Consensus Protocols

The present algorithm correctly replaces consensus protocols. Contrary to other DPU algorithms presented in the thesis, the present algorithm is not an instance of the generic Algorithm 2 (see Section 5.1), since it slightly modifies the additional parameters passed upon service calls and responses. This modification allows us to take advantage of the properties related to those parameters, i.e., the Validity and Agreement properties as defined in Section 2.2.5. Based on those two properties, we are able to guarantee that each instance of consensus is executed on each stack

by the same consensus protocol (which ensures the correctness of our DPU algorithm). This is done as follows. Once a replacement is initiated, the new consensus protocol $newP$ is sent to all stacks with a reliable broadcast protocol. Upon delivery of the new protocol, every stack attaches $newP$ to the proposal value of the following consensus instances. By the Validity property of consensus, the new protocol will be attached to some decision value. The Agreement property ensures that the same new protocol $newP$, if any, is attached to the decision value of each instance of consensus. Thus, $newP$ can be safely installed after the decision.

The description of our replacement algorithm for consensus protocols is structured as follows. We first describe some basic assumptions required by our algorithm and discuss them. We then present the algorithm itself and prove its correctness. It should be noted that because the algorithm is dedicated to consensus protocols only, no characterization is provided.

Basic Assumptions. Each correct stack participates to all replaceable consensus instances.[1] Particularly, for each instance k, every correct stack issues a call k and the corresponding destination set is equal to Π.

Our second assumption guarantees that replaceable consensus instances are initiated sequentially. In other words, on each stack, a call k to replaceable consensus, where k is an integer, occurs only after the response for replaceable consensus instance $k - 1$ has taken place. Note that this restriction can be easily removed, but it complicates the algorithm. For presentation reasons, we chose to discuss the simplest version of our algorithm.

Finally, we consider that an unbounded number of replaceable consensus instances are initiated by protocol stacks. This assumption is only required for termination of dynamic updates (i.e., to ensure the Replacement Termination property). We argue that this hypothesis is realistic for the following reason: If no more consensus instance occurs, then the replacement is useless.

Algorithm 4. The replacement of the current protocol by a new protocol $newP$ is initiated by a call replaceProtocol($newP$) executed at line 6. This call triggers a call to the reliable broadcast service (RBcast) to diffuse the new protocol $newP$ to all stacks. Upon the corresponding response R on some stack i (line 8), we first check at line 9 that protocol $newP$ has not already been bound to the consensus service (i.e., $newP \notin boundProtocols_i$). This test prevents multiple binding of the same protocol if, for instance, the protocol has been locally bound before response R occurs. If the test succeeds, protocol $newP$ is added to the list of protocols to be bound to the consensus service ($replacingProtocols_i$) and the Boolean variable $repRequested_i$ is set to *true* (lines 10-11). From this point on, the replacement algorithm does nothing: it just waits for calls to replaceable consensus (denoted by $r\text{-}consensus$).

[1] As mentioned in Section 4.2.1, protocol stacks call the service provided by the DPU algorithm (i.e., the replaceable consensus service) instead of directly calling the consensus service.

Algorithm 4 DPU Algorithm for Consensus Protocols: code of stack i.

```
 1: Initialisation:
 2:    currentP_i ← initial_protocol_module          {The module currently bind to consensus}
 3:    repRequested_i ← false                          {Is there a replacement requested?}
 4:    replacingProtocols_i ← λ                  {The list of protocols to be bound to consensus}
 5:    boundProtocols_i ← ∅        {The set of protocols that have been already bound to consensus}

 6: upon replaceProtocol(newP) do
 7:    RBcast.call((INTERNAL, newP), Π)

 8: upon RBcast.response((INTERNAL, newP), src) do
 9:    if newP ∉ boundProtocols_i then
10:       add newP to the end replacingProtocols_i
11:       repRequested_i ← true

12: upon r-consensus.call(k, dst, proposal) do
13:    newP ← first element in replacingProtocols_i (or nil if replacingProtocols_i = λ)
14:    consensus.call(k, dst, (proposal, repRequested_i, newP))

15: upon consensus.response(k, src, (decision, replacement, newP)) do
16:    if replacement then
17:       unbind currentP_i
18:       newP_i ← local module of newP
19:       bind newP_i
20:       RBcast.call((removeProt, currentP_i), Π)   {This line is only necessary for protocol removal}
21:       currentP_i ← newP_i
22:       remove newP from the list replacingProtocols_i
23:       boundProtocols_i ← boundProtocols_i ∪ newP
24:       if replacingProtocols_i = λ then
25:          repRequested_i ← false
26:    r-consensus.response(k, src, decision)
```

Whenever the replaceable consensus service is called (line 12), the call is simply redirected to the module currently bound to the consensus service (line 14). Upon this redirection, the $proposal$ value passed as an additional parameter of the service call to $r\text{-}consensus$ is modified to ($proposal$, $replacement$, $newP$).[2] The Boolean value $replacement$ is set according to the value of $repRequested_i$ and indicates to other stacks if a replacement is needed. The variable $newP$ represents the new protocol to be bound to the consensus service and is chosen among the protocols in the list $replacingProtocols_i$ (line 13).

When a consensus module returns a decision (by issuing a response to the consensus service, see line 15), the $replacement$ variable is tested. If its value is *true*, the replacement takes place (lines 16-21). Then, the list $replacingProtocols_i$ and the set $boundProtocols_i$ are updated (lines 22-23). If no more replacement has to be performed (i.e., the list $replacingProtocols_i$ is empty), the Boolean variable $repRequested_i$ is set to *false* (lines 24-25). Finally, for any value of the $replacement$ variable, a response to the replaceable consensus service with the

[2] By definition, consensus protocols solve consensus for a proposal of any type. So the modification of the proposal has no impact on the current consensus protocol. Note that such a modification makes our DPU algorithm incorrect for atomic commitment protocols. The reason is that, contrary to consensus protocols, atomic commitment protocols expect only specific values for the proposal: only *commit* or *abort* are correct proposal values.

value $decision$ as an additional parameter is initiated at line 26.

Note that line 20 is only used for protocol removal (see Section 4.4). Actually, it informs the other stacks that protocol $currentP$ does not issue anymore responses on stack i. If no more response is issued by protocol $currentP$ on every stack, this protocol can be removed. The details about the delivery of the message sent at line 20 can be found in Section 4.4.2.

Proofs. Before showing that Algorithm 4 ensures the specifications of DPU when replacing consensus protocols, let us show the following lemma that simplifies the proofs.

Lemma 1. *For any consensus instance k initiated at line 14, all stacks that execute instance k, execute it with the same protocol.*

Proof. The proof is done by recurrence. By the initial-binding-coherence property (see Section 4.3), the module initially bound to consensus is the same for each stack. Since the local replacement of a consensus module requires to execute consensus, each stack executes the first consensus instance with the same protocol.

Assume now that all stacks that execute instance k of consensus, execute it with the same protocol. We now proof that instance $k + 1$ is also executed with the same protocol by all stacks that execute it. Each stack executes the response corresponding to call (i.e., instance) k before any call $k + 1$ occurs (and more generally before any call $k' > k$ occurs). By the Agreement property of consensus, the decision $(decision, replacement, newP)$ passed upon response k is the same on each stack. By lines 16-26, the module bound to the consensus service after response k has been executed is the same for each stack. Thus, instance $k + 1$ is executed with the same consensus protocol by all stacks that execute it. □

From Lemma 1, the Service Preservation, Replacement Agreement and Replacement Order properties trivially hold for our DPU algorithm when replacing consensus protocols. Let us now prove that our algorithm verifies the last DPU correctness property, namely Replacement Termination.

Replacement Termination: Assume that some correct stack i executes the primitive $replaceProtocol$ with protocol $newP$ as a parameter (line 6). Thus, stack i sends $newP$ to all stacks using the reliable broadcast service (line 7). Because the Strong Reliability and Response Atomicity predicates apply to reliable broadcast (see Table 5.1), each correct stack j eventually delivers protocol $newP$, adds $newP$ to its local list $replacingProtocols_j$, and sets its local variable $repRequested_j$ to *true* (lines 8-11) (*).

Let us finish the proof by contradiction. Assume that stack i never binds module $newP_i$ to the consensus service. Thus, no consensus instance ends on a decision ($-$, $true$, $newP$). By lines 24-25 and (*), the variable $repRequested_j$ is eventually forever true on each correct stack. Stacks that are not correct eventually crash and stop participating to consensus instances. Because there is an unbounded number of consensus instances (see assumptions, page 48) and due to the Validity

and Termination properties of consensus, each correct stack eventually forever decides $(-, true, -)$. By line 22, after each decision, each correct stack j removes a protocol from the list $replacingProtocols_j$. Furthermore, by line 13, this protocol was at the head of the list on some correct stack j. Thus, because new protocols are added at the end of the list (see line 10), $newP$ is eventually at the head of the list $replacingProtocols_j$ on each correct stack j. Thus, there is a consensus instance for which all correct stacks propose a value of type $(-, true, newP)$. Due to the Validity property, there is a consensus instance that decides on $(-, true, newP)$. A contradiction. □

6.2 DPU Algorithm for Local Ordering Protocols

We now present a DPU algorithm that is an instance of the generic Algorithm 2, and inherits from its properties.[3] This new DPU algorithm assumes that the protocols that get replaced ensure the Fifo Order predicate. It allows us to replace protocols that locally order responses such as reliable fifo order and reliable causal order protocols (see the characterization paragraph on page 54).

The basic idea is the following. Once a replacement occurs, each stack executes a final call with the current protocol. From this time on, further calls are executed by the new protocol. However, the responses from the new protocol are executed only after all responses to the final calls issued by correct stacks (faulty stacks are detected using a perfect failure detector). At this point, because the protocols that get replaced ensure the Fifo Order predicate, the old protocol may issue only responses to calls issued on faulty stacks. Such responses are discarded, which means that the responses from the new protocol can be safely executed.

We first describe our algorithm. Then, we present its characterization and prove the correctness of our algorithm. Finally, we discuss some variants of our algorithm.

Algorithm 5. We first describe our algorithm when no replacement occurs. Each time a call to the replaceable service *r-p* occurs (line 9), a *transport* call $((FORWARD, k, callNumber_i), dst)$ is issued to the module currently bound to service p (line 10). The variable $callNumber_i$ identifies the module currently bound to service p. This scenario is shown in stack 1 on the left of Figure 6.1.[4]

The responses from service p are handled at line 11. First, the sequence number sn of the response is compared to the local variable $respNumber_i$ (see line 12). The variable $respNumber_i$ identifies the protocol providing service p that is currently allowed to issue responses. Because no replacement occurs, the sequence number sn is equal to the variable $respNumber_i$. Thus, the response is immediately executed (line 13). In the case of transport responses (which are tagged with

[3]For clarity, we do not formally express the DPU algorithm as an instance of the generic algorithm.

[4]For simplicity, we do not show module bindings in the figures of this section.

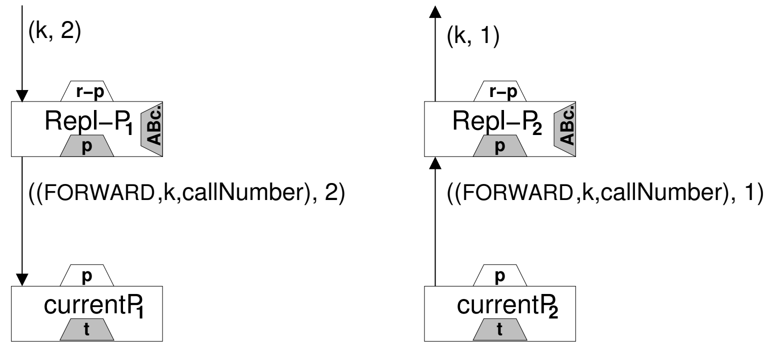

Figure 6.1: Calls (left) and responses (right) when no replacement occurs.

$FORWARD$), a response to service r-p is issued (line 26). This scenario is shown in stack 2 on the right of Figure 6.1.

We now consider the case of a replacement issued on stack i by the call $replaceProtocol(newP)$, where $newP$ is the protocol to be installed (see line 7). Stack i first sends $newP$ to all processes using the atomic broadcast service (ABcast) at line 8.[5] Solid arrows in stack 1 on the left of Figure 6.2 illustrates this. Upon a response from ABcast on stack j (line 16), a *replacement* call $((INTERNAL, newP, callNumber_j), \Pi)$ is issued to the module currently bound to service p (line 17). This is shown in Figure 6.2 by dashed arrows. After issuing the replacement call, stack j locally updates the current protocol module $currentP_j$ with the new protocol module $newP_j$ (line 18-22). The role of line 21 is to allow correct protocol removal (see lines 32-33). Finally, the variable $callNumber_j$ is incremented (see line 23) to correspond to the new protocol executing calls to service p on stack j.

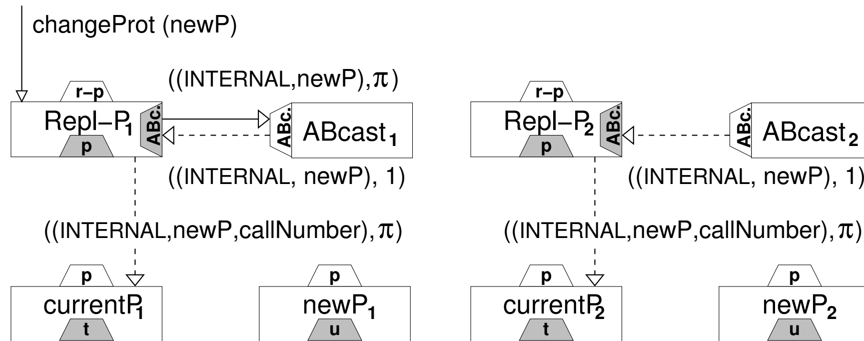

Figure 6.2: The beginning of the replacement on the initiator stack (left) and any other stack (right).

[5] We assume that the modules bound to ABcast do not require service r-p. Otherwise, the DPU may block. Note that this problem does not occur with the other DPU protocols presented in the thesis.

Algorithm 5 DPU Algorithm for Local Ordering Protocols: code of stack i.

```
 1: Initialisation:
 2:    currentP_i ← initial_ protocol_module          {The current protocol used to process calls}
 3:    oldProtocols_i ← λ                          {The lists of protocols that can be later removed}
 4:    undelivered_i[ ] ← [λ, ...]      {The array that contains the lists of responses not yet delivered;
                            one list for each module providing p; all lists are initially empty (λ)}
 5:    callNumber_i = 0                 {The sequence number identifying the protocol used for calls}
 6:    respNumber_i = 0             {The sequence number identifying the protocol used for responses}

 7: upon replaceProtocol(newP) do
 8:    ABcast.call((INTERNAL, newP), Π)

 9: upon r-p.call(k, dst) do
10:    p.call((FORWARD, k, callNumber_i), dst)

11: upon p.response((tag, obj, sn), src) do
12:    if (sn = respNumber_i) then
13:       execute_response(tag, obj, src)
14:    else if (sn > respNumber_i) then
15:       add (tag, obj, src) to the end of list undelivered_i[sn]

16: upon ABcast.response((INTERNAL, newP), j) do
17:    p.call((INTERNAL, newP, callNumber_i), Π)
18:    currentP_i.unbind()
19:    newP_i ← local module of newP
20:    newP_i.bind()
21:    add currentP_i to the end of the list oldProtocols_i
22:    currentP_i ← newP_i
23:    callNumber_i ← callNumber_i + 1

24: procedure execute_response(tag, obj, src)
25:    if (tag = FORWARD) then
26:       r-p.response(obj, src)                                    {Execute transport responses}
27:    else
28:       if execute response(INTERNAL, newP, src) from all non-crashed stacks src ∈ Π then
29:          respNumber_i ← respNumber_i + 1                    {Execute replacement responses}
30:          for all (tag, obj, src) ∈ undelivered_i[respNumber_i] according to the order in the list do
31:             execute_response(tag, obj, src)
32:          oldP ← remove the first element from the list oldProtocols_i
33:          RBcast.call((removeProt, oldP), Π)
```

At this point, each call C to service p is executed by the protocol $newP$. However, the corresponding responses on a stack j corresponding to C are not directly executed in line 13. The reason is that in contrast to the variable $callNumber_i$ on stack i, the variable $respNumber_j$ on stack j is not yet incremented and does not correspond to the new protocol. Thus, responses issued at this time by module $newP_j$ on stack j are added to the list $undelivered_j[sn]$ (see lines 14-15), where sn corresponds to the value of $callNumber_i$ at the moment of call C. Since responses are added to the end of the list $undelivered_j[sn]$, the order of responses is maintained. In Figure 6.3, solid arrows with a black head show the execution of a call to service r-p when module $newP_2$ was recently bound but the variable $respNumber_2$ was not yet incremented.

Finally, upon execution of the replacement responses ($INTERNAL$, $newP$, $callNumber_i$) from all non-crashed stacks (see lines 28-31 and dashed arrows

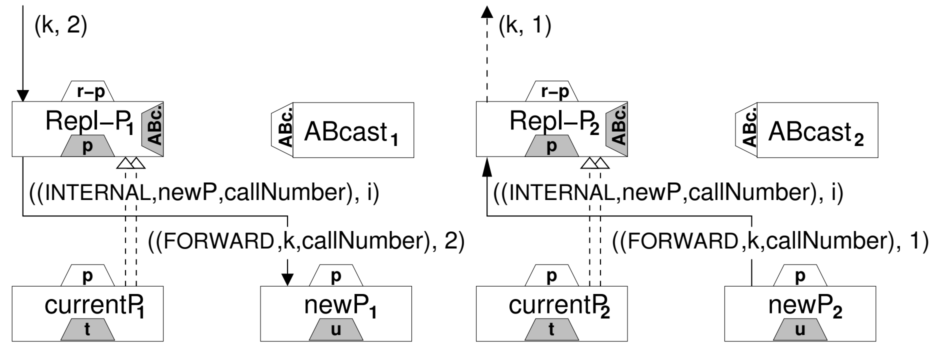

Figure 6.3: The end of the replacement on the initiator stack (left) and any other stack (right).

with a white head in Figure 6.3), the variable $respNumber_i$ is incremented. At this point, the module $newP_i$ is allowed to issue responses. However, all responses from this module that were previously delayed are first executed (see lines 30-31 and the dashed arrow with a black head in Figure 6.3). Note that the responses are executed in the order in which they were inserted in the list $undelivered_i[respNumber_i]$. The last two lines 32-33 implement correct protocol removal. It can be easily observed that the protocol $oldP$ (see line 32) is identified by $respNumber_i - 1$. Thus, all further responses from protocol $oldP$ are discarded on stack i. As a result, the message for protocol removal ($removeProt$, $oldP$) can be safely sent to other stacks (see Section 4.4.2).

Characterization. For identical reasons as Algorithm 3 in Section 5.3.3, Algorithm 5 ensures Replacement Agreement, Replacement Order and Replacement Termination for whatever service p that gets replaced. This is expressed with Rules A, B and C in Table 6.1.

As explained in Section 5.3, the Rules 1-4 in Table 6.1 describe for which service p the algorithm ensures Service Preservation. Specifically, the table shows which predicates apply on service r-p given a service p that gets replaced. For

Rule A:	*true*	⇒	Replacement Agreement
Rule B:	*true*	⇒	Replacement Order
Rule C:	*true*	⇒	Replacement Termination
Rule 1:	At-Most-One-Resp(p)	⇒	At-Most-One-Resp(r-p)
Rule 2:	Fifo Order(p) ∧ Strong Reliability(p)	⇒	Strong Reliability(r-p)
Rule 3:	Fifo Order(p)	⇒	Fifo Order(r-p)
Rule 4:	Causal Order(p)	⇒	Causal Order(r-p)

Table 6.1: Characterization of Algorithm 5.

instance, consider the reliable fifo order service on which the following service predicates apply (see Table 5.1): At-Most-One-Response, Weak Reliability, Strong Reliability and Fifo Order. By Rule 2 in Table 6.1, service r-p is strongly reliable. Thus, because Strong Reliability is strictly stronger than Weak Reliability, service r-p is also weakly reliable. Moreover, by the Rules 1 and 3 in Table 6.1, service r-p ensures At-Most-One-Response and Fifo Order. Thus, our algorithm correctly replaces reliable fifo order protocols. For similar reasons, it also correctly replaces reliable causal order protocols.

Proofs. The proofs that Algorithm 5 ensures the properties Replacement Agreement, Replacement Order and Replacement Termination for any service p that gets replaced are similar to those in Section 5.3.3. For this reason, these proofs are omitted here. We now focus on the proofs for the Service Preservation property, which consists in showing that Algorithm 5 ensures Rules 1-4 of Table 6.1. Let us first state several observations that allow us to simplify the proofs.

Observation 1. *Because Replacement Order holds, a protocol can be unambiguously identified by sequence number sn.*

Observation 2. *Because of lines 17-22, the last call executed on each stack by any module bound to service p is a replacement call.*

Observation 3. *Once a protocol sn is allowed to issue responses on stack i (i.e., $respNumber_i$=sn), all responses from this protocol that were delayed (i.e., the responses in the list $undelivered_i[sn]$) are executed in the order in which they were issued (see lines 14-15 and 30-31). Afterwards, new responses from protocol sn are executed in the order in which they are issued (see lines 11-13). Thus, the order of responses issued by a given protocol sn is maintained by the DPU algorithm.*

Observation 4. *From lines 11-15 and the fact that variable $respNumber_i$ is always increased, we have the following property. If $sn' > sn$, all responses from service r-p that result from a response issued by a protocol sn are issued before responses from service r-p that result from a response issued by a protocol sn'.*

In the proof that follows, we adopt the following simplified notation. A transport call (($FORWARD$, k, sn), dst) to service p on stack i is denoted by $TC_{p,i}(k, sn, dst)$. Because of Observation 1, we use the sequence number sn to identify the protocol executing the call rather than the local variable $callNumber_i$. A transport response (($FORWARD$, k, sn), src) from service p is denoted by $TR_{p,i}(k, sn, src)$. Similarly, replacement calls and responses are denoted by $RC_{p,i}(newP, sn, dst)$ and $RR_{p,i}(newP, sn, src)$.

Rule 1: At-Most-One-Response(p) $\Rightarrow$ At-Most-One-Response(r-p): Consider a call $C_{r\text{-}p,i}(k, dst)$ that occurs on some stack i (line 9). Then, a transport call $TC_{p,i}(k, sn, dst)$ is executed by the protocol sn (line 10). From the predicate At-Most-One-Response of service p, we know that at most one transport response $TR_{p,j}(k, sn, src)$ occurs on each stack $j \in dst$ (*).

Upon transport response $TR_{p,j}(k, sn, src)$ on each stack j (see lines 11-15), three cases are possible:

- $sn < respNumber_j$: Nothing is done.
- $sn = respNumber_j$: The response is immediately executed (line 13) and one single response $R_{r\text{-}p,j}(k, src)$ is issued at line 26.
- $sn > respNumber_j$: The tuple ($FORWARD$, k, src) is added to $undelivered_j[sn]$ (line 15). From lines 30-31, we know that all responses in $undelivered_j[sn]$ are executed at most once (line 31). Thus, because at most one tuple ($FORWARD$, k, src) is in $undelivered_j[sn]$ and no other transport response $TR_{p,j}(k, sn, src)$ occurs on stack j due to (*), at most one response $R_{r\text{-}p,j}(k, src)$ is issued at line 26. □

Rule 2: Fifo Order(p) $\wedge$ Strong Reliability(p) $\Rightarrow$ Strong Reliability(r-p): Consider a call $C_{r\text{-}p,i}(k, dst)$ that occurs on correct stack i (line 9). Then, a transport call $TC_{p,i}(k, sn, dst)$ is issued (line 10) and executed by the protocol sn (and thus, $callNumber_i$=sn). From Strong Reliability of service p, a transport response $TR_{p,j}(k, sn, src)$ is eventually executed on every correct stack $j \in dst$ (line 11). Upon response $TR_{p,j}(k, sn, src)$ on stack j, we distinguish three cases:

- $sn < respNumber_j$: The last call executed by protocol sn on stack i is a replacement call $RC_{p,i}(newP, sn, \Pi)$ (see Observation 2). Because service p is fifo ordered, the transport response $TR_{p,j}(k, sn, src)$ is executed before the replacement response $RR_{p,j}(newP, sn, i)$ on stack j. Since the variable $respNumber_j$ is increased after reception of the replacement response (see lines 28-29), this case is not possible.
- $sn = respNumber_j$: The response is immediately executed (line 13) and a response $R_{r\text{-}p,j}(k, src)$ is issued at line 26.
- $sn > respNumber_j$: The tuple ($FORWARD$, k, src) is added to $undelivered_j[sn]$ (line 15). Because $callNumber_i$=sn, process i has executed line 23 sn times. Thus, process i has executed sn responses from ABcast (line 16). Due to the properties of ABcast, each correct stack $l \in \Pi$ eventually executes sn responses from ABcast and issues sn replacement calls. From Strong Reliability of service p, stack j eventually executes sn replacement responses from every correct stack $l \in \Pi$ and increases the variable $respNumber_j$ until it reaches sn. Then, the tuple ($FORWARD$, k, src) is executed (lines 30-31) and a response $R_{r\text{-}p,j}(k, src)$ is issued (line 26). □

Finally, the following two results follow trivially from Observations 3 and 4:

Rule 3: Fifo Order(p) $\Rightarrow$ Fifo Order(r-p).

Rule 4: Causal Order(p) $\Rightarrow$ Causal Order(r-p).

First Variant. We now briefly explain how Algorithm 5 can be modified in order to ensure an additional characterization rule (see below). More precisely, the modification allows us also to correctly replace best effort fifo protocols.

Rule 5: Fifo Order(p) $\wedge$ Weak Reliability(p) $\Rightarrow$ Weak Reliability(r-p)

In order to understand our modification, let us first explain why Algorithm 5 does not ensure this rule. If service p is weakly reliable, this implies that replacement responses may be lost. Thus, variable $respNumber_i$ may never be incremented to correspond to the new protocol (see lines 28-29). As a result, the responses initiated by the new protocol are never executed and no more response from service r-p are initiated by the DPU algorithm. This clearly violates weak reliability of service r-p. This problem can be easily solved by modifying line 17 to issue an infinite number of replacement calls for each new protocol to be installed. Thus, because service p is weakly reliable, at least one replacement response per correct stack and new protocol occurs on each correct stack.

Second Variant. We now discuss how to make our algorithm independent from the detection of non-crashed stacks (with a perfect failure detector), which restricts the scope of applicability of our DPU algorithm. More precisely, Rule 4 no more holds with this second variant.

The basic idea of our second variant is to consider on each stack i an array of variables $respNumber_i[n]$ instead of a single variable $respNumber_i$. The jth element in the array denotes the protocol allowed to issue responses that correspond to a call issued on stack j. This implies the following modifications to Algorithm 5:

- **Lines 11-15.** A response is executed if the corresponding call was issued on stack j with the protocol identified by sn, such that $sn = respNumber_i[j]$. Otherwise, the response is added to the end of the list $undelivered_i[sn]$.
- **Lines 28-31.** Each variable $respNumber_i[j]$ is incremented upon reception of a replacement response from stack j. At the same time, we execute all responses in the list $undelivered_j[respNumber_i[j]]$ that correspond to a call issued on stack j (i.e., $src = j$).

With these modifications, Observation 4 no more holds. The reason is that the elements of the array $respNumber_i[n]$ are incremented independently from each other. Thus, different protocols may be allowed to issue responses at the same time on a given stack i. As Observation 4 no more holds, we can no longer prove Rule 4. Note that Rule 3 still holds, since our variant ensures the following property. On each stack i, a response issued by protocol sn is executed before a response issued by protocol sn' if $sn' > sn$ and both responses correspond to calls issued on the same stack.

6.3 DPU Algorithm for Global Ordering Protocols

In the previous section, we described a DPU algorithm for local ordering protocols. This algorithm requires a perfect failure detector to detect the non-crashed stacks (see line 28 of Algorithm 5). In contrast, this section describes a DPU algorithm that (1) does not require perfect failure detection, and (2) is dedicated to protocols that ensure a global order of responses (i.e., protocols that ensure at least the Generic Order predicate), such as generic broadcast and atomic broadcast protocols.

The central idea of our algorithm is the following. Each time a replacement is initiated, a special call is initiated to the service p provided by the module that gets replaced. This special call conflicts with all other calls to service p. Since this algorithm assumes that p ensures the Generic Order predicate, the responses to the special calls are totally ordered with respect to the other responses issued from the module that gets replaced. This information allows us to synchronize the local replacement on all stacks.

The remainder of this section is structured as follows. In order to facilitate the understanding of the present algorithm, we first present its main algorithmic differences with Algorithm 5. We then describe preliminary assumptions that simplify our algorithm for global ordering protocols and provide a detailed description of it. Finally, we present its characterization and prove its correctness.

Differences with Algorithm 5. There are three major differences between the DPU algorithm for local ordering protocols (i.e., Algorithm 5) and the one for global ordering protocols:

1. **Diffusion of the new protocol.** Algorithm 5 diffuses the new protocol using atomic broadcast (see line 8), while the present algorithm uses the service p that gets replaced. As a result, the Replacement Order, Replacement Agreement, and Replacement Termination do not hold for any service p with the latter algorithm (see characterization paragraph on page 61).

2. **Replacement steps.** In Algorithm 5, the replacement requires two steps in addition to the diffusion of the new protocol. First, upon the reception of the new protocol $newP$, a replacement call is issued to the old protocol (lines 16-17). From this time on, the calls to the replaceable service are redirected to $newP$ (see lines 18-23), but only the responses from the old protocol are allowed. Second, upon the reception of all responses corresponding to the replacement calls issued by correct stacks, the responses from $newP$ are allowed and those from the old protocol are discarded (lines 28-31).

   In contrast, the present algorithm merges these two steps. More precisely, upon reception of the new protocol $newP$, the calls to the replaceable service are redirected to $newP$ and the responses from $newP$ are allowed. From this time on, all responses from the old protocol are discarded.

3. **Reissuing calls.** Because Algorithm 5 assumes that the service that gets replaced ensures Fifo Order, no response to a call issued by a correct stack to the old protocol is discarded. This is not the case for the present algorithm. Hence, the calls issued to the old protocol by a correct stack for which the responses have been discarded must be reissued to the new protocol.

Preliminary Assumptions. We assume that each call to the replaceable service r-p has a set of destination stacks dst equal to Π. This hypothesis simplifies the algorithm and facilitates its understanding. Note that only minor modifications are required to lift this assumption.

We also consider that a non-infinite number of replacements occurs; actually, we only require that there are sufficient long periods where no replacement occurs. This assumption is necessary, among others, to preserve Strong Reliability of the service that gets replaced (see proofs below). Indeed, upon replacement, some responses from the old protocol may be discarded. In this case, the corresponding calls are reissued to the new protocol. However, if a new replacement occurs, the responses to the reissued calls may be again discarded. Thus, without this assumption, such scenario may occur infinitely.

Algorithm 6. Let us first explain how the algorithm executes a call to the replaceable service r-p (see line 9). Such a call results in a *transport* call to the service p provided by the protocol that gets replaced. The transport call is identified by $(FORWARD, k, seqNumber_i)$: The variable k represents the identifier of the call to r-p, while $seqNumber_i$ identifies locally the protocol used to execute the call to p. Note that contrary to Algorithm 5, this algorithm uses the same variable to identify the protocol used to (1) execute calls and (2) issue responses to/from p. Finally, the call to p is added to the end of the list $pending_i$ that contains the calls to p for which no corresponding response has been yet *effectively* executed (line 11). We say that a response R is effectively executed if R is executed in procedure execute_response (see lines 17-32). We will see later how responses from service p are effectively executed.

The same occurs upon initiation of the replacement of the current protocol by a new protocol $newP$. Such a replacement is initiated by calling the procedure replaceProtocol($newP$) at line 6. The call results in a *replacement* call to service p that is identified by $(INTERNAL, newP, seqNumber_i)$. Note that the replacement call conflicts with all other calls redirected to service p. Again, this call is added to the end of the list $pending_i$ (line 8).

Upon a replacement or a transport response R from service p (executed at line 12), the algorithm first checks the sequence number sn attached to R (sn identifies the protocol that executes the call corresponding to R). If the sequence number corresponds to the protocol currently allowed to issue responses, the response R is effectively executed (see lines 13-14). Otherwise, if (1) sn is bigger than $seqNumber_i$ and (2) there is no replacement response with sequence num-

Algorithm 6 DPU Algorithm for Global Ordering Protocols: code of stack i.

1: Initialisation:
2: $currentP_i \leftarrow$ initial_protocol_module *{The current protocol used to process calls}*
3: $undelivered_i[\,] \leftarrow [\lambda, ...]$ *{The array that contains the lists of responses not yet delivered; one list for each module providing p; all lists are initially empty (λ)}*
4: $pending_i \leftarrow \lambda$ *{The list of pending calls}*
5: $seqNumber_i = 0$ *{The sequence number identifying the protocol used for calls and responses}*

6: **upon** replaceProtocol($newP$) **do**
7: p.call(($INTERNAL, newP, seqNumber_i$), Π)
8: add ($INTERNAL, newP, \Pi$) to the end of list $pending_i$

9: **upon** r-p.call(k, dst) **do**
10: p.call(($FORWARD, k, seqNumber_i$), dst)
11: add ($FORWARD, k, dst$) to the end of list $pending_i$

12: **upon** p.response((tag, obj, sn), src) **do**
13: **if** $sn = seqNumber_i$ **then**
14: execute_response(tag, obj, src)
15: **else if** $sn > seqNumber_i$ **and** $(INTERNAL, -, -) \notin undelivered_i[sn]$ **then**
16: add (tag, obj, src) to the end of list $undelivered_i[sn]$

17: **procedure** execute_response(tag, obj, src)
18: **if** $(tag, obj, -) \in pending_i$ **then**
19: remove $(tag, k, -)$ from list $pending_i$
20: **if** $tag = FORWARD$ **then**
21: r-p.response(k, src); *{Execute transport responses}*
22: **else**
23: $currentP_i.unbind()$; *{Execute replacement responses}*
24: $newP_i \leftarrow$ local module of obj
25: $newP_i$.bind()
26: RBcast.call(($removeProt, currentP_i$), Π)
27: $currentP_i \leftarrow newP_i$
28: $seqNumber_i \leftarrow seqNumber_i + 1$
29: **for all** $(tag, obj, dst) \in pending_i$ according to the order in the list **do**
30: p.call(($tag, obj, seqNumber_i$), dst)
31: **for all** $(tag, obj, src) \in undelivered_i[seqNumber_i]$ according to the order in the list **do**
32: execute_response(tag, obj, src)

ber sn, then response R is delayed and added to the end of the list $undelivered_i[sn]$ (see lines 15-16). In all other cases, response R is discarded.

To complete the description of our algorithm, let us now describe how responses to service p are effectively executed (see lines 17-32). First the call corresponding to the response is removed from the list $pending_i$. In the case of transport responses, a response to service r-p is simply issued at line 21. Otherwise, the replacement is performed as follows. First, the current module is locally updated with a module of the new protocol (lines 23-27). Similarly to the previous algorithms, line 26 is necessary for protocol removal. Then, the sequence number is incremented to correspond to the new protocol (line 28) and all calls in $pending_i$ are reissued to the new protocol (lines 29-30). Finally, similarly to Algorithm 5, the responses from the new protocol that were previously delayed are now executed (lines 31-32).

Rule A:	Gen. Broad.(p)	$\Rightarrow$	Replacement Agreement
Rule B:	Gen. Broad.(p)	$\Rightarrow$	Replacement Order
Rule C:	Gen. Broad.(p)	$\Rightarrow$	Replacement Termination
Rule 1:	Gen. Broad.(p)	$\Rightarrow$	At-Most-One-Resp.(r-p)
Rule 2:	Gen. Broad.(p)	$\Rightarrow$	Strong Reliability(r-p)
Rule 3:	Gen. Broad.(p)	$\Rightarrow$	Resp. Atomicity(r-p)
Rule 4:	Gen. Broad.(p) $\wedge$ Fifo Order(p)	$\Rightarrow$	Fifo Order(r-p)
Rule 5:	Gen. Broad.(p) $\wedge$ Causal Order(p)	$\Rightarrow$	Causal Order(r-p)
Rule 6:	Gen. Broad.(p)	$\Rightarrow$	Gen. Order(r-p)
Rule 7:	Gen. Broad.(p) $\wedge$ Total Order(p)	$\Rightarrow$	Total Order(r-p)

Table 6.2: Characterization of Algorithm 6.

Characterization. Because Algorithm 6 uses the service p provided by the module that gets replaced to diffuse the new protocol to other stacks (line 7), the Replacement Agreement, Replacement Order, and Replacement Termination properties are not ensured independently from service p. More precisely, these properties hold for services that ensure the *Generic Broadcast* predicate (Gen. Broad.) as described by Rules A, B, and C in Table 6.2. The Generic Broadcast predicate is introduced here for readability and is the conjunction of the following four predicates: At-Most-One-Response, Strong Reliability, Response Atomicity and Generic Order.

Let us now illustrate Table 6.2 when the service p that gets replaced is the atomic broadcast service. This service ensures the following six predicates: At-Most-One-Response, Weak Reliability, Strong Reliability, Response Atomicity, Generic Order and Total Order (see Table 5.1). By definition, the Generic Broadcast predicate holds for the atomic broadcast service. From Rules A, B and C, the Replacement Agreement, Replacement Order and Replacement Termination properties are ensured for any service p that ensures the Generic Broadcast predicate (which is the case of the atomic broadcast service).

We now consider the Service Preservation property. From Rule 2, the predicate Strong Reliability applies to the replaceable service r-p when the service p that gets replaced is atomic broadcast. Because Strong Reliability is strictly stronger than Weak Reliability, the latter predicate also applies to service r-p. Similarly, from Rules 1, 3, 6, 7, the other predicates that apply to atomic broadcast also apply to the replaceable service. As a result, the Service Preservation property holds for the atomic broadcast service. Therefore, we can conclude that our DPU algorithm correctly replaces atomic broadcast protocols. For similar reasons, our DPU algorithm is correct for generic order and fifo atomic broadcast protocols.

Proofs. Let us first recall that the Generic Broadcast predicate is the conjunction of the At-Most-One-Response, Strong Reliability, Response Atomicity and Generic Order predicates. In order to simplify the proofs, we start with the following lemma, which allows us to identify a protocol by the sequence number sn on each stack i. We then state some observations that simplify the following proofs.

Lemma 2. *Consider that service p ensures the Generic Broadcast predicate. For each sequence number sn, we have the following property. If some stack i identifies protocol P by sequence number sn, then each correct stack j identifies protocol P by sn.*

Proof. By the initial-binding-coherence property (see Section 4.3), the module initially bound to service p is the same for each stack i. Thus, the same protocol P is identified by $seqNumber_i = 0$ on each stack i (*).

Consider now that some stack i identifies protocol $newP$ by sequence number $sn = 1$. By lines 17-28, this means that stack i effectively executes a replacement response R with $obj = newP$ when $seqNumber_i = 0$. By line 12-16 and because $seqNumber_i$ is only increased upon replacement response, replacement response R is the first replacement response issued by protocol P on stack i. Because service p ensures the Response Atomicity predicate, response R is issued by P on each correct stack j. By the Generic Order predicate and the fact that a replacement response conflicts with all other responses, response R is the first replacement response issued by protocol P on each correct stack j. Thus, by lines 13-16 and because the $seqNumber_j$ is incremented upon execution of a replacement response, replacement response R is the first replacement response effectively executed on each correct stack j. As a result, each correct stack j identifies the same protocol $newP$ with sequence number $seqNumber_j = 1$.

By a simple induction, we can conclude that the same result holds for any sequence number $sn > 0$. □

Observation 5. *Once a protocol sn is allowed to issue responses on stack i (i.e., $seqNumber_i$=sn), all responses from this protocol that were delayed (i.e., the responses in the list $undelivered_i[sn]$) are effectively executed in the order in which they were issued (see lines 15-16 and 31-32). Afterwards, new responses from protocol sn are effectively executed in the order in which they are issued (see lines 12-14). Thus, the order of responses issued by a given protocol sn is maintained by the DPU algorithm.*

Observation 6. *From lines 13-16 and the fact that variable $seqNumber_i$ is always increased, no response from protocol sn is effectively executed after a response from protocol sn' if $sn' > sn$.*

Observation 7. *Because calls to r-p are directly redirected to the protocol that is currently bound to p (see lines 9-10) and reissued to a new protocol in the order in which they occur (see lines 11 and 29-30), the order of calls issued to the replaceable service r-p is maintained by the DPU algorithm.*

Observation 8. *Assume that service p ensures the Generic Broadcast predicate. Consider a stack i that identifies a protocol by $sn \geq 1$. Because the replacement response that results in identifying protocol sn conflicts with all other responses, the responses issued by protocol $sn - 1$ that are effectively executed on any stack j is a subset of those that are effectively executed on stack i.*

By Lemma 2, Rules A and B hold trivially.

Rule A: Gen. Broad.(p) $\Rightarrow$ Replacement Agreement.

Rule B: Gen. Broad.(p) $\Rightarrow$ Replacement Order.

We now prove the remaining rules.

Rule C: Gen. Broad.(p) $\Rightarrow$ Replacement Termination: Assume that some correct stack i calls the primitive replaceProtocol($newP$) at line 6. Thus, the replacement call C is added to the end of the list $pending_i$ (line 8). We finish the proof by contradiction: Assume that module $newP_i$ is never bound to service p on stack i. By lines 18-27, call C is forever in the set $pending_i$ (*).

We now consider the time when all faulty stacks have been crashed and no more replacement is initiated (see preliminary assumptions, page 59). Let us denote by sn the protocol allowed to execute calls and to issue responses to/from p on stack i at this time. Because of lines 29-30 executed after protocol sn has been bound to p and due to (*), call C has been (re)issued to protocol sn. Let us denote by R the first replacement response that is issued by protocol sn on stack i. Since p ensures the Strong Reliability predicate, such a response eventually occurs. By lines 13-16 and because the $seqNumber_i$ is only incremented upon execution of R, response R is effectively executed and stack i identifies a protocol P by $sn + 1$. By Lemma 2, each correct stack j identifies protocol P by $sn + 1$. Thus, stack j also effectively executes response R. By lines 18-19, the replacement call that corresponds to response R is removed from the set $pending_j$ of the stack j that issues the replacement call (**).

Because no more replacement is initiated, the only replacement calls issued to protocol $sn + 1$ are initiated from the set $pending_j$ of every correct stack j at lines 29-30 (upon the effective execution of response R). By (**), the number of replacement calls reissued to protocol $sn + 1$ is strictly smaller than the number of replacement calls that have been issued to protocol sn.

By a simple induction, we can show that stack i eventually identifies a protocol sn' such that no replacement call is reissued to sn'. A contradiction with (*). $\square$

Rule 1: Gen. Broad.(p) $\Rightarrow$ At-Most-One-Resp.(r-p): Assume that some stack i issues a call C to service r-p at line 9. Thus, a transport call C' is added to the list $pending_i$ (line 11). Consider that a response to C occurs on stack j. By lines 17-21, only the responses to C' that are effectively executed result in a response to C. We identify by R' the first response to C' that is effectively executed on some stack j and by sn the protocol that issues R'.

We now show that no other response to C' are effectively executed on stack j. Because service p ensures the At-Most-One-Response predicate, no other response to call C' is issued by sn on stack j. By Observation 6, no response to C' issued by a protocol $sn' < sn$ is effectively executed. For the protocols $sn' > sn$, the following two cases have to be considered:

- **Stack i does not identify any protocol $sn' > sn$.** As a result, stack i does not reissue call C' to a protocol $sn' > sn$. Thus, no response to call C' is issued by any protocol $sn' > sn$ on stack j.
- **Stack i identifies a protocol $sn' > sn$.** By Observation 8, stack i has effectively executed the response R' issued by protocol sn. By lines 18-19, the call C' has been removed from the list $pending_i$. Thus, no further response R' is issued by a protocol $sn' > sn$ on stack j. □

Rule 2: Gen. Broad.(p) ⇒ Strong Reliability(r-p): Assume that a call C to service r-p occurs on some stack i at line 9. Thus, a transport call C' is added to the list $pending_i$ (line 11). Because there is a limited number of replacement, there is a protocol (that we identify by sequence number sn) that is the last protocol to be bound to service p on stack i. By Lemma 2, protocol sn is the last protocol identified by each correct stack. Two cases have to be considered:

- **There is a protocol $sn' < sn$ such that stack i effectively executes a response to call C' issued by sn'.** By Observation 8, each correct stack also effectively executes a response to call C' issued by sn'. By lines 17-21, a response to call C is issued on each correct stack.
- **There is no protocol $sn' < sn$ such that stack i effectively executes a response to call C' issued by sn'.** By lines 17-21, call C' has not been removed from $pending_i$ before stack i identifies protocol sn. By lines 29-30, call C' is reissued to protocol sn. Because the service p ensures the Strong Reliability predicate, protocol sn eventually issues a response R' to call C' on each correct stack. Because the variable $seqNumber_j$ is no more incremented after a correct stack j identifies protocol sn (otherwise, sn is not the last protocol to be bound to service p on stack j), each correct stack effectively executes the response R'. By lines 17-21, a response to call C is issued on each correct stack. □

Rule 3: Gen. Broad.(p) ⇒ Response Atomicity(r-p): Consider a call C to service r-p. Assume that some response R to C occurs on stack i. By lines 17-21, a transport response R' has been effectively executed on stack i. Let us identify by sn the protocol that issues R'. Consider now any correct stack $j \neq i$. By Lemma 2, stack j identifies the same protocol by sn. The following two cases have to be considered:

- **Stack j identifies a protocol by $sn' > sn$.** By Observation 8, stack j also effectively executes the response R' issued by sn. By lines 17-21, a response to call C is issued on stack j.

- **Stack j does not identify any protocol by $sn' > sn$.** Because service p ensures the Response Atomicity predicate, a transport response R' is issued by protocol sn on stack j. Because the variable $seqNumber_i$ is no more incremented after stack j has identified protocol sn (otherwise, stack j identifies a protocol by $sn' > sn$), stack j effectively executes the response R'. By lines 17-21, a response to call C is issued on stack j. $\Box$

Finally, the following four results trivially hold by Observations 5 to 8.

Rule 4: Gen. Broad.(p) $\wedge$ Fifo Order(p) $\Rightarrow$ Fifo Order(r-p).

Rule 5: Gen. Broad.(p) $\wedge$ Causal Order(p) $\Rightarrow$ Causal Order(r-p).

Rule 6: Gen. Broad.(p) $\Rightarrow$ Gen. Order(r-p).

Rule 7: Gen. Broad.(p) $\wedge$ Total Order(p) $\Rightarrow$ Total Order(r-p).

6.4 Related Work

We present in this section two published DPU algorithms. The first algorithm, the *Switching Algorithm*, has been described in [BKvRC01] and has been designed for the Ensemble [Ens01, RBH$^+$98] group communication toolkit. The *Adaptive Total Order Algorithm* described in [MR06] has been implemented in the Appia [App01, MPR01] protocol framework. These two algorithms can be implemented with our approach (described in Chapters 4 and 5). Other DPU algorithms have been described in literature. However, most of these algorithms require to modify the protocols that get replaced. Examples are given in Section 4.2.2.

Switching Algorithm [BKvRC01]. To our understanding, the switching algorithm behaves like Algorithm 5, except during the switching phase.[6] Contrary to Algorithm 5, the switching algorithm assumes that the destination set of each call is equal to Π. Moreover, it is not clear how the switching algorithm tolerates failures. We now describe the switching algorithm by pointing out the main conceptual differences with Algorithm 5:

- Instead of issuing a replacement call to the protocol sn that must be replaced (see line 17 of Algorithm 5), each stack i sends to all stacks the number of calls nc_i that have been executed locally by protocol sn.
- Instead of incrementing the variable $respNumber_i$ when all responses to replacement calls occur on stack i (see lines 28-29 of Algorithm 5), this variable is incremented when the number of responses nr_i issued by protocol sn on stack i corresponds to the number of calls executed by protocol sn on all (non-crashed) stacks. In other words, the variable $respNumber_i$ is incremented when $nr_i = \sum_{j=1}^{n} nc_j$.

[6] No formal description of this algorithm has been found in literature.

To our understanding, the switching algorithm correctly replaces similar protocols as Algorithm 5. However, the lack of description of the mechanism to tolerate failures makes it difficult to understand which protocols are correctly replaced by the switching algorithm.

Adaptive Total Order Algorithm [MR06]. The central idea of this algorithm is to use both the protocol that gets replaced and the new protocol during the switching phase. According to the authors, this allows to reduce the impact of the DPU algorithm during the replacement. This claim has to be compared with the results that we obtained in Section 6.5.3.

We now describe how the adaptive total order algorithm behaves upon replacement. Upon initiation of a replacement on some stack i, stack i diffuses the new protocol to all stacks. Once a stack j delivers the new protocol $newP$, stack j issues a special call to the protocol P that gets replaced. From this time on, each call issued to the replaceable service is redirected to both (1) the protocol that gets replaced and (2) the new protocol. However, only the responses from protocol P are executed. The responses issued by $newP$ are simply delayed.

Once the responses corresponding to the special calls issued by all non-crashed stacks occur on some stack j, all responses from the old protocol are discarded. Furthermore, all the responses from the new protocol (including those that were delayed) are executed. However, before executing a response R from the new protocol, the algorithm checks that no similar response has been already issued by the old protocol. The check ensures that for each call redirected to the old and the new protocols, only one response is executed.

We can make the following comments. First, the algorithm assumes that replacement are initiated sequentially, i.e., a new replacement can be initiated only if no replacement currently occurs. Second, the algorithm is dedicated to fifo atomic broadcast protocols only. Finally, similarly to Algorithm 5, the adaptive total order algorithm requires a perfect failure detector. Note that the authors explain how to modify their algorithm so that it only requires a $\diamond S$ failure detector. However, no convincing argument is provided to show that the algorithm is still correct with $\diamond S$.

6.5 Performance Evaluation

The goal of this section is to evaluate the overhead induced by DPU algorithms when replacing consensus or fifo atomic broadcast protocols. The overhead is computed in the following two cases: (1) when no replacement occurs, and (2) during the replacement. Our experiments shows that the overhead induced by our DPU algorithms is acceptable (around 30% in most cases) when no replacement occurs. On the other hand, during the replacement, the overhead significantly increases (it can reach 800%). However, it should be noted that the effects of the replacement are observed only during a short period.

We start this section by defining our experimental setup. Specifically, we define the system setup, the benchmark and the performance metrics considered in our experiments. Then, we present our results when replacing consensus protocols. Finally, we evaluate several algorithms to replace (fifo) atomic broadcast protocols, and compare their performance.

6.5.1 Experimental Setup

System Setup. The benchmarks were run on a cluster of machines running SuSE Linux (kernel 2.6.11). Each machine has a Pentium 4 processor at 3.2 GHz and 1 GB of RAM. The machines are interconnected by Gigabit Ethernet (which is exclusively used by the cluster machines) and run Sun's 1.5.0 Java Virtual Machine, but use a light-weight marshaling library [PHN00, HNMP05] instead of standard Java serialization [Sun04]. The machines were dedicated to the performance benchmarks and had no other load on them.

Benchmark. Our experiments use the group communication middleware described in Section 4.2.1 (see Figure 4.3). The overhead of DPU algorithms is computed on the basis of the performance of the (fifo) atomic broadcast service when using either (1) a DPU algorithm for consensus[7] or (2) a DPU algorithm for fifo atomic broadcast protocols. Note that the atomic broadcast algorithm of our group communication middleware also provides the *fifo* atomic broadcast service. In order to evaluate independently the overhead of each DPU algorithm, the DPU module that implements replacement of consensus protocols is removed from the stack when the overhead of the DPU algorithm for (fifo) atomic broadcast protocols is evaluated, and vice versa.

Each of our experiment has been conducted as follows. Messages of a given size were sent using the (fifo) atomic broadcast service under a constant load on each stack. In the middle of the experiment, some process triggers a (single) replacement and then continues to issue messages to the (fifo) atomic broadcast service. Note that the replacement procedure updates a protocol P with a new protocol $newP$, such that P and $newP$ implement exactly the same algorithm. This allows us to measure the exact impact of the replacement. We consider that the replacement starts when some process triggers a replacement, and finishes when all stacks have replaced the old modules by new modules.

For each DPU algorithm that we tested, several experiments were conducted involving either $n = 3$ or $n = 7$ stacks (machines). Furthermore, we varied (1) the size s of messages sent, and (2) the offered load l under which the messages were sent. The *offered load* specifies the maximal amount of messages that are sent per second among all stacks. We chose a simple symmetric workload where all processes send messages at the same load. Note that our atomic broadcast

[7] Since our atomic broadcast protocol uses consensus, the performance of atomic broadcast is influenced by the replacement of consensus protocols.

algorithm uses a flow-control mechanism that blocks a process when too many messages are sent. The flow-control mechanism has been designed so that the performance of the atomic broadcast protocol is optimized.

For each value of the parameters n, s and l and each DPU algorithms, several experiments were conducted. We also conduct for each value of the parameters n, s and l, some experiments without any DPU algorithm in our stack. The performance obtained with these last experiments were used as a reference to observe the overhead induced by DPU algorithms.

We have only evaluated the overhead of DPU algorithms in good runs, i.e., without any process failures. This overhead is measured once the system has reached a stationary state (at a sufficiently long time after the start). We ensure that the system stays in a stationary state by verifying that the performance of atomic broadcast stabilizes over time.

Performance Metrics. We use two performance metrics to evaluate the performance of atomic broadcast: *average latency* and *throughput* [Urb03]. For a single message sent using atomic broadcast, the average latency L is defined as follows. Let t_0 be the time at which the message is sent and let t_i be the time at which the message is delivered on stack i, with $i \in 1, \ldots, n$. The average latency L is then defined as $L \stackrel{def}{=} (\sum_{i=1}^{n} t_i - t_0)/n$.

Two other measures for latency can be considered: (1) *early latency* (i.e., $min_i\ (t_i - t_0)$) and (2) *late latency* (i.e., $max_i\ (t_i - t_0)$). The results for these two measures are similar to the results for average latency, and thus, will not be discussed.

The throughput T is defined as follows. Let r_i be the rate at which messages are delivered on a stack i, with $i \in 1, \ldots, n$. The throughput T is then defined as $T \stackrel{def}{=} \frac{1}{n} \sum_{i=1}^{n} r_i$ and is expressed in messages per second (or msgs/s).

In our performance evaluation, the mean for L and T is computed over many messages and for several executions. For all results, we computed 95% confidence intervals. Note that for most experiments the confidence intervals are negligible and are not shown in this case.

6.5.2 Replacement of Consensus Protocols

We now present the results that we obtained when using Algorithm 4 (see Section 6.1), i.e., our algorithm for replacement of consensus protocols. We first show that the impact of the replacement for specific values of the parameters n, s and l. We then present the overhead as a function of these parameters.

Figure 6.4 shows the impact of one replacement. The figure shows the average latency as a function of the time at which the corresponding message was sent using atomic broadcast. We show the results of several experiments with the same parameters, which is why several values are shown on the vertical axis for a given time t. This allows us to make the impact of the replacement clearly visible. The

experiments are with 3 stacks. In each experiment, the replacement was invoked at time 5000. The offered load and the message size are respectively equal to 4000 msgs/s and 4KB. We can observe that the average latency increases around $t = 5000$, but quickly stabilizes to reach the level it had before the replacement.

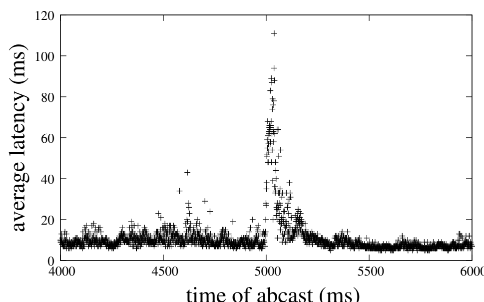

Figure 6.4: Latency as a function of the time at which ABcast is issued.

Figure 6.5 shows both the average latency (top) and the throughput (bottom) as a function of the offered load (left) or message size (right). For the graphs where the offered load varies, the message size is set to 4KB, and for the graphs where the message size varies, the offered load is set to 1000 msgs/s. Note that we only show results for experiments involving 3 stacks. The results obtained with 7 stacks are similar. The dashed graphs represent the results obtained with the normal version of our group communication stack, i.e., without any DPU algorithm. The solid graphs show the results obtained when no replacement occurs but with our DPU algorithm for consensus. Finally, the dotted graphs represent the results during the replacement. Figure 6.5 shows that the cost of adding a DPU algorithm for consensus protocols is acceptable (around 30% in most cases). It also shows that the overhead induced by our DPU algorithm during the replacement is important (it can reach 800%). However, it should be noted that this overhead is observed only during a short period: less than one second (see Figure 6.4).

6.5.3 Replacement of Fifo Atomic Broadcast Protocols

Similarly to our DPU algorithm for consensus, our DPU algorithm for fifo atomic broadcast protocols induce (1) an acceptable overhead when no replacement occurs, and (2) an important overhead during the replacement (which lasts for a short period). For this reason, we do not show the corresponding graphs. Instead, we compare two different DPU algorithms that can be used to replace fifo atomic broadcast protocols.

We compare (1) Algorithm 6 (see Section 6.3), and (2) the Adaptive Total Order algorithm described in [MR06] (see Section 6.4).[8] Our comparison shows that Algorithm 6 has the best performance. It should be also noted that, contrary to

[8] In order to fairly compare these two algorithms, we adapt the Adaptive Total Order algorithm, so that it allows replacement to be initiated concurrently.

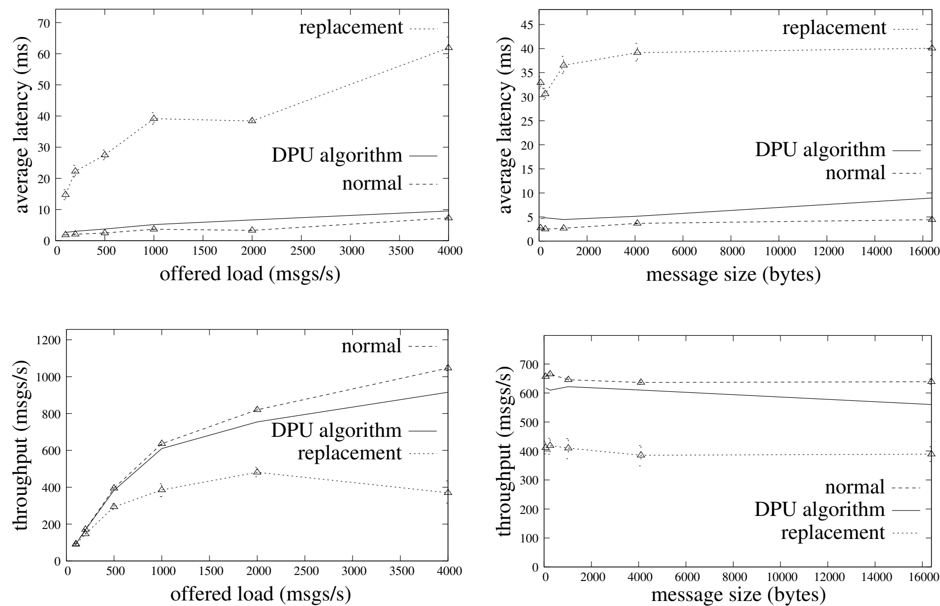

Figure 6.5: Average latency (top) and throughput (bottom) as a function of the offered load (left) or message size (right).

the Adaptive Total Order Algorithm, it allows the replacement of non-fifo atomic broadcast protocols, and does not require a perfect failure detector.

Figures 6.6 and 6.7 show the performance obtained respectively when no replacement takes place and during the replacement. In each figure, both the latency (top) and the throughput (bottom) are represented as a function of the offered load (left) or message size (right). For the graphs where the offered load varies, the message size is set to 4KB, and for the graphs where the message size varies, the offered load is set to 1000 msgs/s. The solid and dashed graphs represent the results obtained respectively with Algorithm 6 and the Adaptive Total Order algorithm [MR06]. The dotted graph shows the results obtained with our stack without any DPU algorithm.

Figures 6.6 and 6.7 clearly show that Algorithm 6 performs better than the Adaptive Total Order algorithm. This can be explained by the fact that contrary to other Algorithm 6, the Adaptive Total Order algorithm uses both the old and the new protocols during the replacement. Furthermore, in order to ensure that each message is delivered only once on each stack, the Adaptive Total Order algorithm requires (1) additional computations, and (2) to attach to each message some information that is not required by our algorithm. Note that these results seem to contradict the results presented in [MR06], where the authors compare the Adaptive Total Order algorithm with Algorithm 6 thanks to the SSFNet network simulator [NLLY03]. However, these differences may be explained by the different system setup considered: a simulated network in [MR06] versus a real network here.

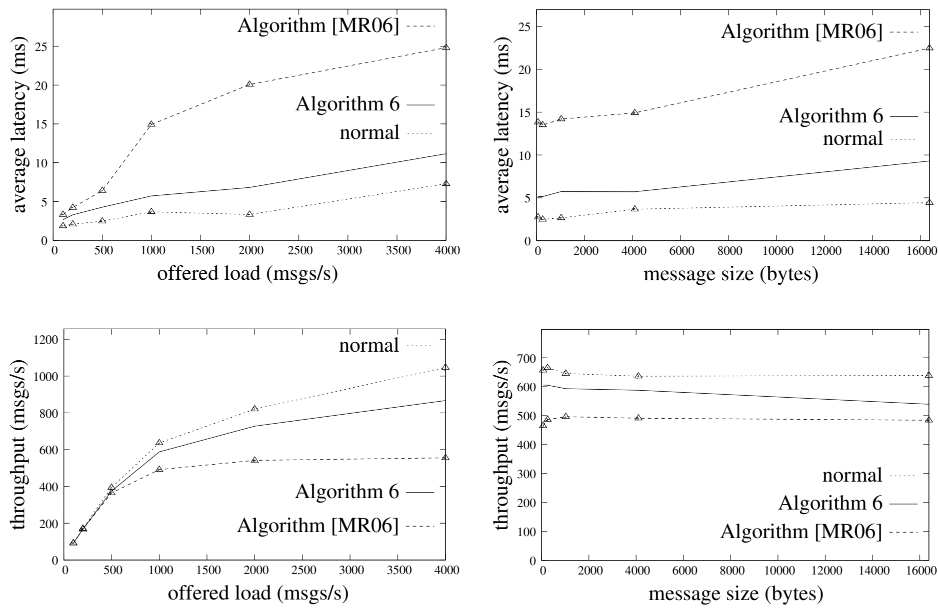

Figure 6.6: Average latency (top) and throughput (bottom) as a function of the offered load (left) or message size (right), when no replacement occurs.

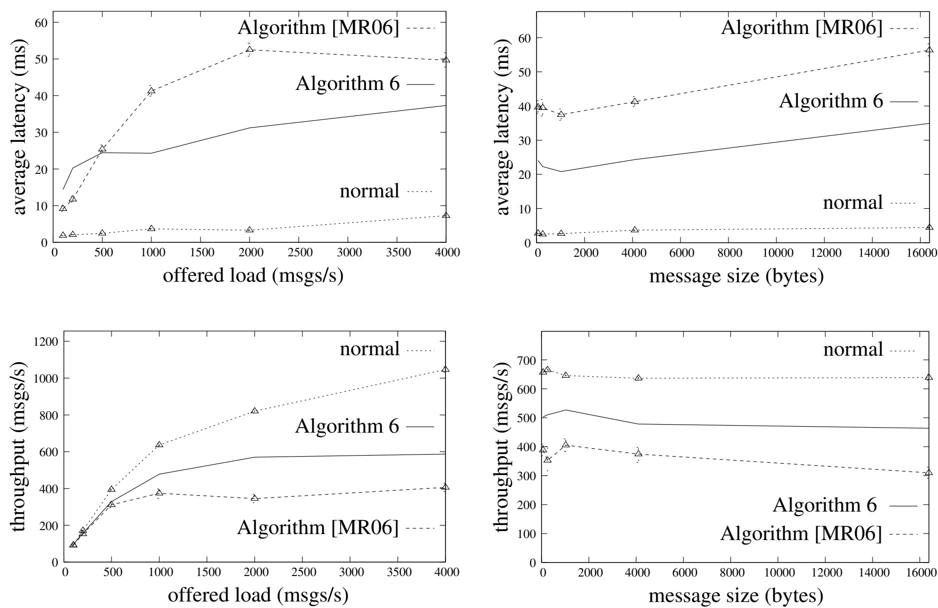

Figure 6.7: Average latency (top) and throughput (bottom) as a function of the offered load (left) or message size (right), during the replacement phase.

6.6 Conclusion

In this chapter, we have first described several DPU algorithms to correctly replace group communication protocols. Specifically, we presented DPU algorithms for (1) consensus protocols, (2) local ordering protocols, and (3) global ordering protocols. All these algorithms allowed us to validate our approach for DPU described in Chapters 4 and 5. Furthermore, we discussed two existing DPU algorithms.

In a second step, we evaluated the overhead induced by DPU algorithms. We have shown that the overhead is acceptable when no replacement occurs. We have also shown that the important overhead during the replacement phase can be tolerated, since the effects of the replacement lasts during a short period of time. Finally, we have shown that our algorithm for global ordering protocols performs well compared to an existing algorithm when replacing fifo atomic broadcast protocols.

Chapter 7

Service Interface: A Convenient Abstraction to Implement Modular and Updateable Distributed Protocols

This chapter focuses on protocol frameworks, which are programming tools for developing modular (and updateable) distributed protocols: protocol frameworks allow complex protocols to be implemented by composing several simple modules that cooperate with each other.

Most protocol frameworks, such as Cactus [Cac01, BHSC98], Appia [App01, MPR01], Ensemble [Ens01, RBH$^+$98], Eva [BGT$^+$01], SDL [SDL00] and Neko [Nek01, UDS02], are based on *events*. Events are simple abstractions that allow protocol modules in a stack to interact with each other. However, the use of events raises some problems (as shown in Section 7.4). For instance, the composition of modules may require connectors to route events, which introduces a burden for a protocol composer [BMN05, EMPS04]. Protocol frameworks such as Appia and Eva extend the event-based approach with channels. This solution is, however, not satisfactory since composition of complex protocol stacks becomes more difficult.

In [BMN05], the authors propose a new approach for implementing distributed protocols, which is based on *message headers* rather than on events. Despite this approach solves most of the problems inherent to the use of events, it is not considered in this chapter since it does not provide the necessary features for the implementation of *updateable* protocols.

In this chapter, we propose a new approach for developing modular and updateable protocols, that is based on *service interfaces*. We compare this new approach with the classical event-based approach. We show that service-interfaces has several advantages, such as (1) adequate representations of protocol module interactions, (2) fairly straightforward compositions of protocol modules, and (3) integrated mechanisms to facilitate implementation of DPU managers (the modules

that implement the DPU algorithms described in Chapter 6). For all these reasons we advocate the use of service interfaces (instead of events) to implement modular (and updateable) distributed protocols. To validate our claim, we implement an experimental framework based on service interfaces, called SAMOA [Sam08]. We use SAMOA to compare the service-interface- and the event-based implementations of the group communication stack illustrated in Figure 4.3 (see Section 4.2).

7.1 Protocol Module Interactions

In this chapter, we consider the three following kinds of protocol module interactions between modules of the same stack. Note that these interactions are very similar to service calls and responses (which were introduced in Chapter 4).

- **Requests** are issued by protocol modules or applications. A request corresponds to an invocation of a (single) module P_i in order to use the service provided by P_i. The result of a request, if any, is not returned upon the end of the invocation, but requires an additional interaction (see Replies).
- **Replies** transport the result of a request. A single request may generate several replies on the same stack or on different stacks. Only the modules of the same protocol as the module that initiates the request are concerned by the corresponding replies. For example, a request by module P_i in stack i generates replies that concern only module P_j in any stack j ($j = i$ or $j \neq i$).
- **Notifications** allow a module to inform (possibly many) other modules about a meaningful change in its state. A notification issued by module P_i can be modeled, for each module Q_i concerned by the notification, as an implicit request to P_i (to require the state of P_i) plus an explicit reply (to return to Q_i the state of P_i).

7.2 Event-Based Protocol Frameworks

In event-based protocol frameworks, protocol modules cooperate thanks to the following two abstractions, namely *events* and *handlers*.

Events. An *event* is a special object for communication between protocol modules in the same stack. Events may transport some information, e.g., a network message or some other data. Each event is an instance of an *event type*. Events enable one-to-many indirect communication, i.e., a module that *triggers* an event (1) interacts with possibly several modules and (2) is not aware of the modules it interacts with.

Handlers. Protocol modules are implemented as sets of *handlers*, which are special methods dedicated to *handle* events. In the same module, handlers may share data. Each handler can be dynamically *bound* to some event type. Handlers can also be *unbound* dynamically.

Upon triggering some event e, all handlers bound to the type of e are executed. If no handler is bound, the behavior is usually unspecified. Triggering an event can be done either synchronously or asynchronously. In the former case, the thread that triggers an event e is blocked until all protocol modules that handle e have finished handling of event e. In the latter case, the thread that triggers the event is not blocked.

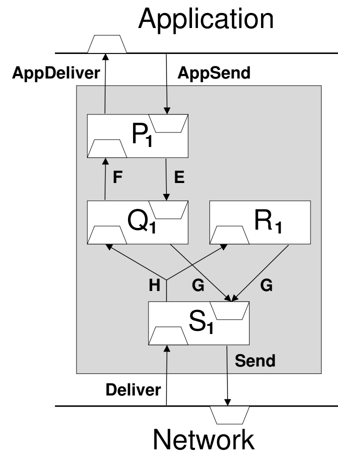

Figure 7.1: Example of an event-based protocol stack (stack 1).

In Figure 7.1, we show an example of an event-based stack. A handler implemented by module P_1 is represented by a white trapezoid inside module P_1. Event types are denoted by capital letters (e.g., E, F and G) and are represented by arrows. An arrow from module P_1 to module Q_1 with label E denotes that (1) module P may trigger events of type E, and (2) module Q implements a handler bound to E. Note that the network and the application are represented as special modules that respectively handle events Send and AppDeliver and respectively trigger events Deliver and AppSend.

Specific Features. Some protocol frameworks have special features to improve the use of events. Below, we present some of those features.

In Cactus [Cac01, BHSC98], the programmer can give a priority number to a handler upon binding it to an event type. When an event is triggered, all handlers are executed following the order of priority. A handler h is also able to *cancel* the execution of an event trigger: all handlers that should be executed after h according to the priority are not executed.

Appia [App01, MPR01] and Eva [BGT$^+$01] introduce the notion of *channels*. Channels allow to build routes of events in protocol stacks. Each protocol module has to *subscribe* to one or many channels. All events are triggered by specifying a channel. When a protocol module triggers an event e specifying channel c, all handlers bound to the type of e, that are part of a protocol that subscribes to c, are executed (in the order prescribed by the definition of channel c).

7.3 Service-Interface-Based Protocol Frameworks

We now describe our new approach for implementing protocols which is based on service interfaces. We show in Section 7.4 the advantages of service-interface-based protocol frameworks over event-based protocol frameworks.

In a service-interface-based framework, protocol modules in the same stack communicate by issuing requests, replies and notifications to a special object called *service interface*. In order to process these protocol module interactions, protocol modules are implemented as a set of special components, called *executers*, *listeners* and *interceptors*, which may share data. We now describe these components.

Executers. *Executers* handle requests. An executer can be dynamically *bound* to a specific service interface. It can be later *unbound*. However, at most one executer at any time can be bound to a service interface on each stack.

A request issued to a service interface si leads to the execution of the executer bound to si. If no executer is bound to si, the request is delayed until some executer is bound to si.

Listeners. *Listeners* handle replies and notifications. Similarly to executers, listeners can be dynamically *bound* and *unbound* to/from a specific service interface si. However, several listeners can be bound to a single service interface.

A *notification* issued to a service interface si is handled by all listeners bound to si in the local stack. On the other hand, a *reply* issued to a service interface si is handled by one single listener (that is not necessarily bound to si). To ensure that one single listener handles a reply, a module P_i has to identify, each time it issues a request, the listener (belonging to P_i) that handles the possible reply. If the request and the reply occur, respectively in stack i and in stack j, the service interface si on i communicates to the service interface si' on j the listener that must handle the reply. If the listener that must handle the reply does not exist, the reply is delayed until the listener is created.

In Figure 7.2, we show an example of a service-interface-based stack. We denote a service interface by small letters (e.g., t, u, v and net) in an hexagonal box. The service interface net allows to access the network and the service interface t allows the application to interact with the stack. The dashed black arrow from module P_1 to service interface u shows that module P_1 may generate requests to u. The fact that module P_1 may generate replies and notifications to service interface t is represented by a dashed white arrow. Similarly, module S_1 may generate (1) requests to service interface net and (2) replies and notifications to service interface v. We represent executers with white boxes inside protocol modules and listeners with white boxes with a gray border. For example, module P_i implements an executer and a listener. The connecting line between service interface t and the executer e implemented by P_i shows that e is bound to t. Similarly, the fact that the listener l implemented by P_i is bound to service interface u is represented by a connecting line between l and u.

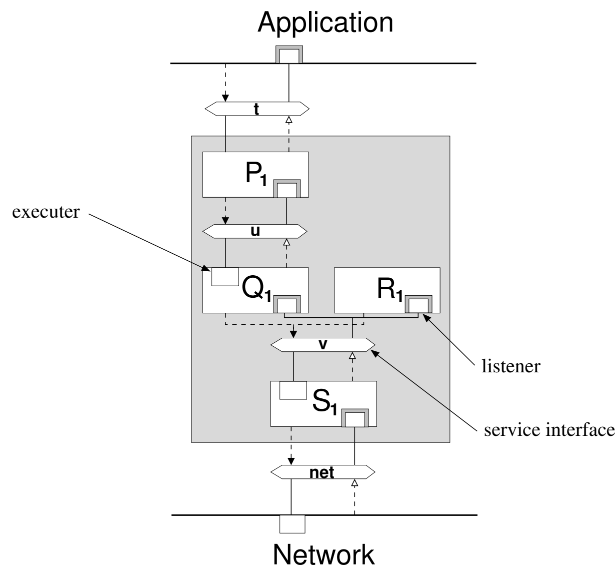

Figure 7.2: Example of a service-interface-based protocol stack (stack 1).

Interceptors. An *interceptor* plays a special rôle. Similarly to executers, interceptors can be dynamically bound or unbound to a service interface. They are executed each time a request, a reply or a notification is issued to the service interface they are bound to. This is illustrated in Figure 7.3. In the right part of the figure, the interceptor of the protocol module T_1 is represented by a rounded box. The interceptor is bound to service interface t. The left part of the figure shows that an interceptor can be seen as an executer plus a listener. When P_1 issues a request req to the service interface t, the executer-interceptor of T_1 is executed. Then, module T_1 may forward a request req' (possibly $req' \neq req$) to the service interface t.[1] When module Q_1 issues a reply or a notification, a similar mechanism is used, except that this time the listener-interceptor of T_1 is executed. In short, a protocol module T_i, that has an interceptor bound to a service interface t, is able to intercept, modify, cancel or delay the requests, replies and notifications that are issued to t.

Upon requests, if several interceptors are bound to the same service interface, they are executed in some deterministic order. Upon replies and notifications, the order is reversed.

[1]The two service interfaces t in the left part of Figure 7.3 represent the *same* service interface t. The duplication is only to make the figure readable.

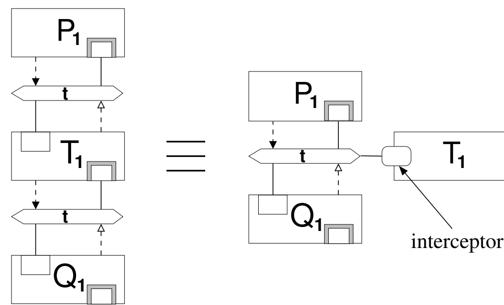

Figure 7.3: Execution of protocol interactions with interceptors (in stack 1).

7.4 Advantages of Service-Interface-Based Protocol Frameworks

This section aims at showing the advantages of protocol implementations based on service interfaces over the ones based on events. We structure our discussion in three parts. First, we present how protocols interactions (as defined in Section 7.1) are modeled in both event-based and service-interface-based protocol frameworks. Then, we discuss the composition of protocol modules. Note that protocol composition actually corresponds to define module bindings (see Section 3.1). Finally, we show that, contrary to events, service interfaces provide a convenient mechanism to implement DPU managers (see Section 4.2). The whole discussion is summarized at the end of the section.

7.4.1 Protocol Module Interactions

For each protocol module interaction, we describe how it is modeled in both categories of frameworks. This leads to the following two conclusions: (1) service interfaces provide an adequate model for each interaction, while (2) events inadequately model requests and replies, which increases the risk of programming errors.

Requests. In service-interface-based frameworks, a request is generated to a service interface. Each request is handled by at the most one executer, since we allow only one executer to be bound to a service interface at a time. On the other hand, in event-based frameworks, a protocol module emulates a request by triggering an event. There is no guarantee that this event is bound to only one handler, which may lead to programming errors (remember that by definition, a request should not be processed by several modules).

Replies. When a protocol module generates a reply to a service interface, only the correct listener (identified when the corresponding request was issued) is executed. This ensures that a request issued by some protocol module Q_i, leads to replies handled only by protocol modules Q_j (i.e., protocol modules of the same protocol).

This is not the case in event-based frameworks, as we now show. Consider protocol module Q_1 in Figure 7.1 that triggers event of type G to emulate a request. Module S_1 handles the request. When modules S_i triggers an event of type H to emulate a reply (remember that a reply can occur in any stack), both modules Q_i and R_i will handle the reply (they both contain a handler bound to H). This behavior is not correct: by definition, only protocol modules Q_i should handle the reply. Moreover, as modules R_i are not necessarily implemented to handle replies dedicated to Q_i, this may lead to errors.

Solutions to solve this problem exist. However, they introduce an unnecessary burden on the protocol programmers and the stack composer. For instance, channels allow to route events to ensure that modules handle only relevant events. However, the protocol programmer must take channels into account when implementing protocols. Moreover, the composition of stacks becomes more difficult due to the fact that the composer has to create many channels to ensure that modules handle events correctly. An addition of special protocol modules (named *connectors* [BMN05, EMPS04]) for routing events is also not satisfactory, since it requires additional work from the composer and introduces overhead.

Notifications. Contrary to requests and replies, notifications are well modeled in event-based frameworks. The reason is that notifications correspond to the one-to-many communication scheme provided by events. In service-interface-based frameworks, notifications are also well modeled. When a module generates a notification to a service interface si, all listeners bound to s are executed.

7.4.2 Protocol Module Composition

Since replies are the results of a request, there is a semantic link between these two kinds of interactions. The composer of protocol modules must preserve this link in order to compose correct stacks. We explain now that service based frameworks provide a mechanism to preserve this link, while in event-based frameworks the lack of such mechanism leads to error-prone composition.

In service-interface-based frameworks, requests and corresponding replies are issued to the same service interface. Thus, a service interface introduces a link between these interactions. To compose a correct stack, the composer has to bind to each service interface si an executer of a module that issues replies to si. Applying this simple methodology, ensures that requests issued to a service interface si are (1) effectively handled by some executer, and (2) result in replies issued to the same service interface si.

In event-based frameworks, all protocol interactions are issued through different event types. As a result, there is no explicit link between an event triggered upon a request and an event triggered upon the corresponding reply. Thus, the composer of a protocol stack must know the exact meaning of each event type in order to preserve the semantic link between replies and requests, and no simple methodology can be applied to compose correct stacks. Moreover, nothing pre-

vents the binding of a handler that should handle requests to an event type used to issue replies. This last problem can be, however, solved by typing handlers.

7.4.3 Implementation of DPU Managers

In Figure 7.4, we show how our solution to integrate a DPU manager (see Section 4.2) can be implemented in protocol frameworks that are based on service interfaces (in the left part of the figure) and on events (in the right part of the figure). The two-sided arrows point to the protocol modules P_1 and $newP_1$ that are switched.

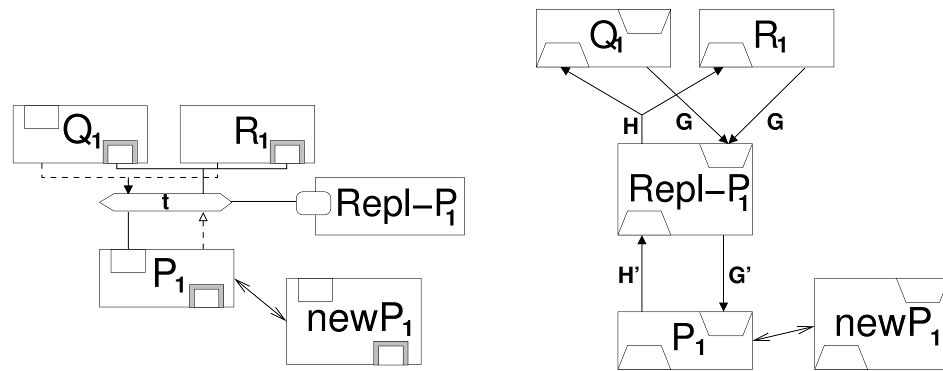

Figure 7.4: Implementation of a DPU manager to replace protocol P (stack 1).

It can be seen that the approach that uses service interfaces has advantages. The intercepting module *Repl*-P_1 has an interceptor bound to service interface t that intercepts every request handled by modules P_1 and all replies and notifications issued by P_1. The addition of a DPU manager in a stack is therefore fully transparent to other protocols.

In event-based frameworks, the solution requires to add an intermediate module *Repl*-P_1 that intercepts the requests issued to P_1 and also the replies and notifications issued by P_1. Although this *ad-hoc* solution may seem similar to the service-interface-based approach, there is an important difference; DPU managers cannot be transparently added to a stack. Indeed, the event-based solution requires to (1) introduce new events (i.e., G' and H') and (2) slightly modify the module P_1 since instead of handling events of type G and triggering event of type H, P_1 must now handle and trigger events of different types G' and H' (see Fig. 7.4).

Note that, contrary to other event-based frameworks, Cactus [Cac01, BHSC98] allows to transparently add DPU managers to protocol stacks. We now explain why. Remember that Cactus allows to define priority for handler executions. This means that handlers with a low priority has a similar behavior as interceptors. Thus, a DPU manager that must intercept some event types G and H can be transparently implemented with two handlers with low priority which are respectively bound to G and H.

7.4.4 Summary

Table 7.1 summarizes the whole discussion.

	service-interface-based	event-based
Protocol Module Interaction	an adequate representation	an inadequate representation
Protocol Module Composition	clear and safe	complex and error-prone
Implementation of DPU Managers	an integrated mechanism	*ad-hoc* solutions

Table 7.1: Service-interface-based *vs.* event-based.

7.5 Implementation

We have implemented an experimental service-interface-based protocol framework, called SAMOA [Sam08]. Our implementation is light-weight: it consists of approximately 1800[2] lines of code in Java 1.5 (with generics).

We now describe the main three classes of our implementation: *Service* (encoding service interfaces), *ProtocolModule* (encoding protocol modules), and *ProtocolStack* (encoding protocol stacks). We next evaluate the service-interface based implementation of a group communication stack with respect to the corresponding event-based stack.

The Service Class. A *Service* object implements two methods: *call* (for issuing requests) and *respond* (for issuing replies and notifications). Both methods accept a special argument, called *message*. A message represents a piece of information sent over the network. Messages determine the kind of interactions issued upon call to method *respond*, as we now explain. When a protocol module executes the method *call*, it has to specify, within the message, the listener that must handle the corresponding reply. When a protocol module executes the method *respond*, a reply is issued if the message passed as an argument specifies a listener. Otherwise, a notification is issued. In addition to a message, each method may accept other arguments. Note that the user have to specify the type of these additional arguments for both methods *call* and *respond* of each *Service* object.

Executers, listeners and interceptors are encoded as inner-classes of the *Service* class. This allows to provide *type-safe* protocol interactions. For instance, executers can only be bound to the *Service* object they belong to. Thus, the parameters passed to requests (that are verified statically) always correspond to the parameters accepted by the corresponding executers.

[2]This number does not take into account the different algorithms for transparent concurrency described in Chapter 8. In total, SAMOA actually consists of approximately 3600 lines of code.

Finally, some *Service* object *so* can be easily defined as a subtype of another *Service* object *so′*. In practice, this means that if a protocol module P_i can issue requests to a service interface *UDP*, then it may also issue a request to a service interface *TCP* that is a subtype of *UDP*.

The ProtocolModule Class. A *ProtocolModule* object consists of three sets of components, one set for each component type (executers, listeners and interceptors). *ProtocolModule* objects are characterized by names to retrieve them easily. Moreover, we have added some features to bind and unbind all executers, listeners or interceptors to/from the corresponding *Service* objects. These simple features facilitate implementation of dynamic replacement of distributed protocols.

The ProtocolStack Class. A *ProtocolStack* object consists of two sets of respectively protocol modules and services. Each *ProtocolStack* object provides a flow control mechanism [Men06] that allows protocol modules in a stack to control the number of requests issued by the application to the stack. We have also implemented methods to add or remove dynamically service interfaces and protocol modules to/from a stack. These methods simplify implementation of protocol addition and removal (see Sections 4.3 and 4.4). Finally, we provide a graphical user interface that facilitates protocol stack composition. Our interface allows the user to store the stack that it has composed as a special file, which can be loaded by *ProtocolStack* objects.

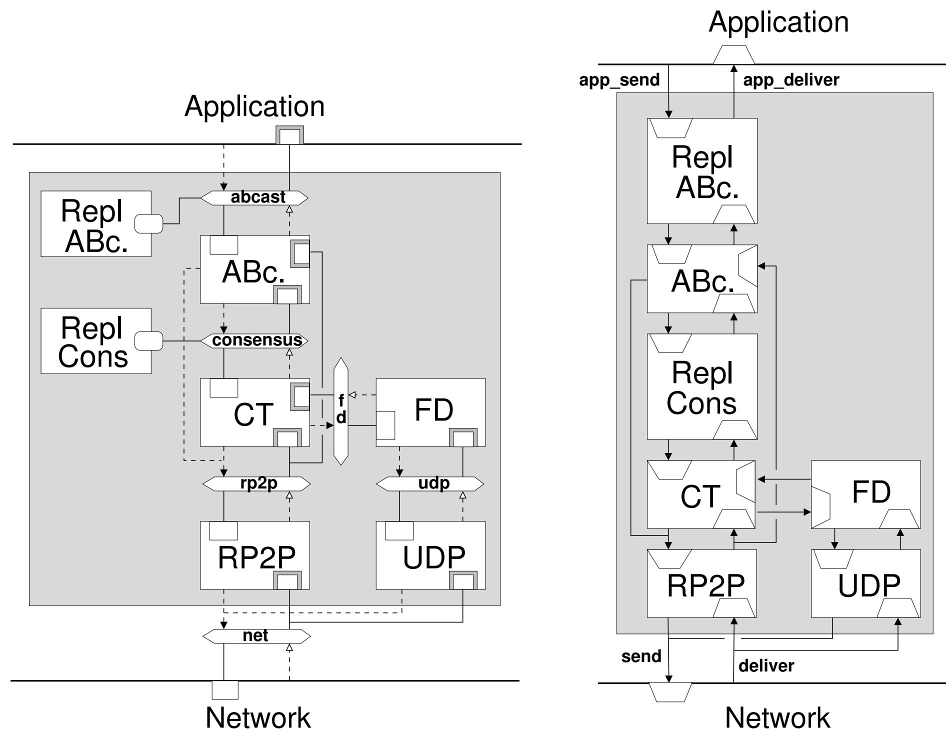

Figure 7.5: Implementation of a group communication stack: service-interface-based (left) *vs.* event-based (right)

Performance Evaluation. To evaluate our implementation of service interfaces, we have implemented the group communication stack illustrated in Figure 4.3 (Section 4.2.1), adopting both the service-interface- and the event-based approaches (see Figure 7.5[3]). The event-based implementation uses the Cactus [Cac01, BHSC98] protocol framework. This choice is not arbitrary: Cactus is widely used and has been shown to have good performance [MCGS03]. Finally, both implementations of our stack assume sequential executions, i.e., only one thread is executed in our protocol stack at a time. Techniques to introduce concurrency in protocol stacks are discussed in Chapter 8.

Performance tests have been performed with 3 and 7 machines (stacks) that send messages of 4KB under a constant offered load using atomic broadcast. All experiments have been performed using the system setup presented in Section 6.5. Similarly, the performance metrics and the benchmark (but without any DPU manager activated) which are considered here are described in the same section.

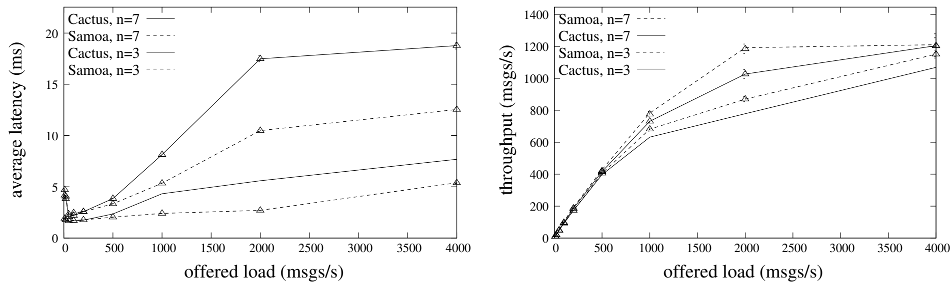

Figure 7.6: Comparison between SAMOA and Cactus: average latency (left) and throughput (right).

In Figure 7.6, we show average latency (left) and throughput (right) both with respect to offered load. Latencies and throughputs are shown on the vertical axis, while offered loads are shown on the horizontal axis. The dashed lines show the results obtained with the SAMOA framework, while the solid lines show the results obtained with the Cactus framework. In all graphs and for all frameworks, the upper (resp. lower) line represents the results obtained with 7 (resp. 3) processes. It can be observed that SAMOA performs better than Cactus. This can be explained by the (necessary) use of connectors in Cactus. Furthermore, the better results obtained by the SAMOA implementation can be explained by a better strategy to implement timeouts: Cactus spawn a new thread for each timeout, while SAMOA manage all timeouts with a single thread (see Section 8.4). Finally, note that similar results were obtained with different message sizes and different metrics (such as early latency or late latency).

[3]Note that each arrow in the event-based architecture represents an event type. We do not name event types in the figure for readability.

7.6 Conclusion

In this chapter, we proposed a new approach for protocol frameworks that is based on *service interfaces* instead of events. We have shown that service-interface-based frameworks have several advantages over event-based frameworks. Service-interface-based frameworks allow us to: (1) model accurately protocol interactions, (2) reduce the risk of errors during the composition phase, and (3) elegantly implement DPU managers. We have finally shown that our experimental protocol framework SAMOA based on service interfaces perform well with respect to Cactus, a representative event-based protocol framework.

Chapter 8

Transparent Concurrency for Modular Protocol Design

We have seen in Chapter 7 a new abstraction that simplifies the implementation of protocol stacks. In this chapter, we discuss techniques to improve the performance of protocol stacks by introducing concurrency. In our opinion, existing protocol frameworks do not provide convenient features to allow concurrent executions in protocol stacks. For example, the Appia protocol framework [App01, MPR01] enforces sequential executions of protocol stacks. On the other hand, the Cactus protocol framework [Cac01, BHSC98] does not restrict the amount of concurrency but it relies on the programmer, who must implement the synchronization to prevent incorrect executions. However, the synchronization code is rather subtle and error-prone, especially in complex protocol stacks, and has to be tailored for the particular stack composition. We propose in this chapter a better solution.

Our solution consists in *transparently* adding concurrency to protocol stacks. In other words, the synchronization mechanisms that allow correct concurrent executions are provided by the runtime system and thus, are hidden from the programmer. We first define in this chapter a correctness property that must be ensured by concurrent executions. Our correctness property, called *module-order*, is very similar to the well-known *isolation* property considered in database systems: it ensures that concurrent request execution is identical to executing the requests sequentially.

We then show how the module-order property can be relaxed in order to increase the amount of concurrency, while ensuring correct executions. Roughly speaking, the relaxation consists in ensuring the module-order property only for the protocol module interactions that require synchronization to be correct. For instance, some synchronization is required for the delivery of atomic broadcast messages (in order to maintain the delivery order), while no synchronization is required for the delivery of reliable broadcast messages (that are not ordered).

We validated our ideas by designing several algorithms to ensure the module-order property. Our algorithms have been implemented within the SAMOA [Sam08] protocol framework that has been described in Section 7.5. This allowed us (1) to

evaluate the amount of concurrency provided by our algorithms, and (2) to compute the gain in performance that we can expect with concurrent executions within a group communication middleware.

8.1 Model

In this section we define a simple model, based on events, for concurrent executions in protocol stacks. We choose the notion of "event" to describe concurrency in protocol stacks for readability. The contributions of this chapter are, however, also relevant to service-interface-based protocol frameworks. Actually, we implemented the algorithms for concurrency that are described in the following sections within the SAMOA [Sam08] protocol framework (which is based on service-interfaces). We describe how to adapt the algorithms to service-interfaces in Section 8.4.

We say that a module P handles some event e if P implements a handler that handles e. We assume that each handler is statically bound to one specific event type. We explain in Section 8.4 how to allow dynamic bindings of handlers, which is required for dynamic protocol update.

Below, we define the notion of computation, which is the basis for our definition of sequential and concurrent executions. Basically, a computation is a set of events that result from a request issued to the stack (i.e., an event triggered externally to the stack). Before formally defining computations, let us introduce some basic notions.

Internal and External Events. An *internal* event denotes an event that has been triggered by any handler implemented by a protocol module in the protocol stack. For instance, events of types E, F, G, H, $Send$ and $AppDeliver$ in Figure 8.1 are internal events. We say that an event is *external* if it is not internal. In Figure 8.1, only events of types $Deliver$ and $AppSend$ are external, since they are triggered respectively by the network and the application.

Causal Dependency. An event e *directly causally depends* on an event f if there is a handler h in the stack that triggers e while handling f. The *causal dependency* relation is defined as the transitive closure of the direct causal dependency relation. Note that each internal event causally depends on exactly one external event, while external events do not causally depend on any event.

Computations. A computation C is the set of all events that causally depend on a specific external event e. We assume that computation C also includes the external event e. Hence, each event (internal or external) is part of exactly one computation.

We show two example computations C and C' in Figure 8.1. Computation C starts with an event of type $AppSend$ and contains also events of type E, G and $Send$. Similarly, computation C' starts with an event of type $Deliver$ and contains also events of types H, F and $AppDeliver$. A dashed arrow from event type E to event type G denotes that, in the same computation, an event of type G directly causally depends on an event of type E.

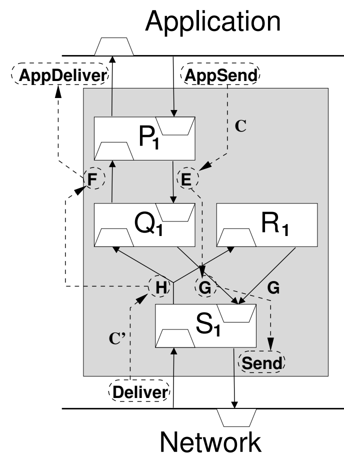

Figure 8.1: Example of computations (in stack 1).

Based on the notions previously introduced, we now define our execution model which precisely defines how computations are executed. We then define sequential and concurrent executions.

Execution Model. We assume that one handler per module is executed at a time.[1] In other words, we do not consider here the problem of concurrent modifications to the data maintained by modules. We say that an event e is handled before an event f if no handler executes f before all handlers executing e have terminated. We assume that events of the same computation C are executed sequentially, i.e., for any pairs of events e and f of C, event e is handled before f or vice versa. In short, we only focus on the issues related to the concurrent executions of events (1) belonging to different computations, and (2) handled by different modules.

A computation starts when the corresponding external event is triggered. At this time, a task (i.e., a thread) dedicated to the execution of the computation is created. Each computation is identified by a unique identifier. We denote by $C(e)$ the identifier of the computation that contains event e.

Each task dedicated to some computation collaborates with a *scheduler*, which ensures the correctness of the execution. The scheduler has also to manage the event triggering initiated by (1) externalTrigger(e) (for an external event e), and (2) internalTrigger(e) (for an internal event e). The task dedicated to a computation is created by the scheduler upon a call to externalTrigger. Upon creation of the computation task, an identifier id and the corresponding external event e are passed as a parameter by the scheduler. In the following sections, we describe schedulers that ensure correct concurrent executions.

[1]This can be easily implemented with a monitor for each module. Note that there is no risk of deadlock since, in our model, no thread can enter a monitor while in another monitor.

We now precisely describe how tasks execute computations (see Algorithm 7). Basically, each task executes sequentially the events of a computation (see lines 4-8). Before executing each event e, the task calls method startHandling(e) (line 5), which blocks until event e can be safely executed (according to the correctness property that is ensured by the scheduler). After the event e is executed, the thread indicates to the scheduler that the event e has been executed by calling the method endHandling(e) (line 7). The next event to be executed is returned by the call to method next(id), where id identifies the computation executed by the thread. The list of events to be executed is updated by the scheduler upon (1) calls to internalTrigger (initiated upon execution of line 6) which adds events to the list, and (2) calls to endHandling which removes events from the list. If all events of the computation have been executed, the method next(id) returns nil.

Algorithm 7 Task for the Execution of a Computation

```
1: task computation(identifier, initialEvent)
2:   id ← identifier                                   {The identifier of the computation}
3:   e ← initialEvent                   {The next event of the computation to be executed}
4:   while e ≠ nil do
5:     startHandling(e)
6:     execute sequentially all handlers bound to the type of e  {The events triggered upon the execution
                                                   of e are later returned by the method next.}
7:     endHandling(e)
8:     e ← next(id)
```

It should be noted that the contribution presented in this chapter can be easily adapted to other thread models. For instance, our schedulers can be adapted to a thread model in which a given thread executes all events handled by a given protocol instead of executing all events of a given computation. Different thread models for protocol stacks have been compared in [MY98].

Sequential Executions. Roughly speaking, an execution is sequential if computations are executed one by one. Formally, an execution is sequential if the following property holds. For any two events e and f that belong to computations C and C' respectively, if C starts before C', then event e is handled before f.

If an execution is not sequential, we say that the execution is *concurrent*. We define in the next sections different ordering properties for concurrent executions.

8.2 The Module-Order Property

This section first defines the *module-order* property, which ensures that the result of a concurrent execution is the same as a particular sequential execution. Note that it is not sufficient that the result of a concurrent execution is equivalent to any sequential execution. For instance, consider two external events triggered by the application to send a message through a fifo protocol. In this case, the external events must be executed in the order in which they are triggered, otherwise the

properties ensured by the fifo protocol are violated. This means that the concurrent execution of these two events must have the same result as the sequential execution in which these two external events occur in the same order. Since no module shares data (see Section 3.1), this can be guaranteed if each module handles events in the same order as in the sequential execution. This is precisely what is ensured by the module-order property, which is formally defined below. We next propose a simple scheduler that ensures the module-order property and prove its correctness. Finally, we present two variants of our simple scheduler that increase concurrency.

Module-Order. Consider two events e and f such that e belongs to a computation that starts before the computation that contains f. If there is a module P that handles both e and f, then e is handled before f.

We now describe a simple scheduler that ensures the module-order property. The basic idea is to order the computations that contain an event handled by a given module P with a list dedicated to P. The computations in the list are ordered according to the order in which they are initiated. When all events in the computation have been handled, the computation is removed from the list. Thus, to ensure the module-order property, an event of a given computation C handled by P can be executed only when computation C is at the head of the list.

Simple Module-Order Scheduler. Algorithm 8 describes our simple module-order scheduler. It assumes that, upon triggering an external event e, the set of modules that may handle an event that belongs to the computation initiated by e is passed as a parameter of the method externalTrigger (see parameter dep in line 5). The set dep can be easily computed by the scheduler if for each handler h, the programmer specifies the type of the events that may be triggered by h. Indeed, from this information, we can deduce the type of all events that may causally depend on a given external event, and thus deduce the modules in the set dep.

The simple module-order scheduler maintains two arrays of lists, namely $modules$ and $computations$. The list $modules[P]$ orders the computations that may contain an event handled by module P according to the order of their initiation. On the other hand, the list $computations[id]$ contains the events of the computation identified by id that have not yet been executed. We now describe how the scheduler ensures the module-order property by describing the following procedures:[2]

- **externalTrigger(**e, dep**)** first adds the identifier $nextId$ corresponding to the computation C initiated by e to the end of the list $modules[P]$ for each module P in the set dep (lines 6-7). Then, the new task that executes computation C is initiated (line 8) and the event e is added to the list of events of C (line 9). Finally, the identifier $nextId$ is incremented to correspond to the next computation (line 10).

[2]Each **procedure** block is executed in mutual execution. A **wait** statement (see line 17) releases mutual exclusion (similarly to a *wait* in a monitor).

Algorithm 8 Simple Scheduler for the Module-Order Property

1: Initialisation:
2: $modules[\,] \leftarrow [\lambda, ...]$ *{The lists of computations that may execute a given protocol module; one list per protocol module; all lists are initially empty (λ)}*
3: $computations[\,] \leftarrow [\lambda, ...]$ *{The lists of events that are triggered but not handled; one list per computation; all lists are initially empty (λ)}*
4: $nextId \leftarrow 1$ *{The identifier for the next computation to be initiated}*

5: **procedure** externalTrigger(e, dep)
6: **for all** $P \in dep$ **do**
7: add $nextId$ to the end of the list $modules[P]$
8: new task computation($nextId$, e)
9: add e to the head of the list $computations[nextId]$
10: $nextId \leftarrow nextId + 1$

11: **procedure** internalTrigger(e)
12: add e to the end of the list $computations[C(e)]$

13: **procedure** next(id)
14: **return** an event e such that $e \in computations[id]$ (nil if $computations[id] = \lambda$)

15: **procedure** startHandling(e)
16: **for all** modules P such that P handles event e **do**
17: **wait** until $C(e)$ is at the head of list $modules[P]$

18: **procedure** endHandling(e)
19: remove e from the list $computations[C(e)]$
20: **if** $computations[C(e)] = \lambda$ **then** *{The computation corresponding to e terminated}*
21: **for all** modules P **do**
22: **if** $C(e) \in modules[P]$ **then**
23: remove $C(e)$ from the list $modules[P]$

- **internalTrigger(**e**)** simply adds the event e to the list of events of the computation (lines 11-12).
- **next(**id**)** returns any event of the computation identified by id (lines 13-14). The order in which the events of the same computation id are returned is not relevant to this chapter, and thus, no specific order is considered here. Note that we discuss this issue in Section 8.4.2.
- **startHandling(**e**)** blocks until $C(e)$ is at the head of all lists of modules that handle e (lines 15-17).
- **endHandling(**e**)** simply removes the event e from the list of events that have not been executed (line 19). Furthermore, when the computation of event e terminates (i.e., the list $computations[C(e)]$ is empty), the identifier $C(e)$ is removed from all lists $modules[P]$ that contain $C(e)$ (lines 20-23).

We now prove that our simple scheduler ensures the module-order property. In addition, we show that our scheduler is *live* in the sense that all computations eventually terminate. Liveness holds under the assumptions that (1) each execution of a handler eventually terminates, and (2) each computation contains a bounded number of events. In other words, liveness is ensured if the code written by the stack programmer is live.

Module-Order: Consider two events e and f such that e belongs to a computation C that starts before the computation C' that contains f. Moreover, assume that there is a module P that handles both events e and f. Because C starts before C', the identifier of computation C is added before the identifier of computation C' in the list $modules[P]$ at line 7 (*).

By procedure internalTrigger (lines 11-12) and definition of computations, the set $computations[C(e)]$ is not empty before the call endHandling(e) terminates. By lines 20-23, the identifier $C(e)$ is removed from the list $modules[P]$ only by this call (more precisely, by the similar call corresponding to the last event of computation $C(e)$), and thus, after event e has been handled (**).

By the definition of the task that executes computations, event f is handled only after the call startHandling(f) returns. Due to lines 15-17, and (*), (**), the call returns after event e has been handled. Thus, event e is handled before event f. □

Liveness: By lines 6-7, the first computation C to start has the following property. For each event $e \in C$, the call startHandling(e) is non-blocking (*). Because of Algorithm 7, and due to our assumptions for liveness (see above), the list $computations[id]$ where id is the identifier of C is eventually empty (i.e., computation C has terminated). Thus, by lines 20-23, the identifier of C is eventually removed from all lists $modules[P]$ that contains this identifier. At this time, the property (*) holds for the computation that was the second to start. By applying a simple induction, we can show that all computations eventually terminate. □

One can observe that our simple scheduler has the following problem: The identifier of a computation C is removed from a list $modules[P]$ only when the computation terminates, instead of removing the identifier when all events of C handled by P have been executed. The next two variants try to detect when no more events from a computation C are handled by a module P. As a result, the identifier of C can be removed earlier from list $modules[P]$, which increases concurrency in the protocol stack.

Our first variant, named *bounded module-order scheduler*, requires to know for each computation C the upper bound b for the number of events that are handled by a given module P. This allows to remove the identifier of C from list $module[P]$ as soon as b events from C have been handled by P. Note that the bound b must be carefully set. Indeed, if the bound b is too large, it may never be reached. In this case, this variant is equivalent to the simple module-order scheduler.

The *route module-order scheduler* – our second variant – is based on the knowledge of the routes in our protocol stack. A route for event type E describes the types of the events that causally depend on events of type E. With this information, we can detect, for a given computation, when all events of a given type have been executed. Thus, a computation C can be removed from the list $module[P]$ as soon as all events of any types to which a handler implemented by P is bound have been executed.

Similarly to the simple module-order scheduler, the information required by our variants (i.e., the bounds or the routes) can be easily computed by the scheduler.

We now describe the modifications to the simple scheduler that are required by our variants. Note that the correctness proofs for these variants are very similar to the ones for our simple scheduler, and thus are omitted here.

Bounded Module-Order Scheduler. Contrary to the simple scheduler, the bounded scheduler assumes that the parameter dep contains a *list* of modules, where each module P appears as many times as the maximum number of events of the corresponding computation that P may handle. In addition, the following two procedures have to be modified:

- **externalTrigger(**e, dep**)** adds b times the computation identifier corresponding to e to the end of the list $modules[P]$ for each module P, where b corresponds to the number of occurrence of P in dep. Lines 8-10 remain the same.
- **endHandling(**e**)** removes one occurrence of $C(e)$ from the list $modules[P]$ for each module P that handles event e. When the computation terminates, all occurrences of the computation identifier are removed from all lists $modules[P]$.

Route Module-Order Scheduler. Contrary to the previous variant, the route scheduler assumes that the parameter dep is a directed graph, where each node is an event type and each arrow between two event types E and F shows that an event of type F might be triggered upon handling an event of type E. Moreover, the following two procedures have to be adapted:

- **externalTrigger(**e, dep**)** adds the computation identifier corresponding to e to the end of the list $modules[P]$ for each module P that implements a handler bound to a node of dep that can be reached from the type of e. Lines 8-10 remain the same.
- **endHandling(**e**)** removes $C(e)$ from the list $modules[P]$ for a module P if the following condition holds: Every event in the list $computations[id]$ has a type E such that all handlers implemented by P are bound to a type that is not reachable from E in dep. When the computation terminates, the computation identifier $C(e)$ is removed from all lists $modules[P]$ that contain this identifier.

8.3 Relaxing the Module-Order Property

We first present in this section a simple example showing that the module-order property is too restrictive in the sense that it prevents several concurrent executions that are actually correct. Based on this observation, we define a new property, called *relaxed module-order* that allows more concurrent executions. Finally, we explain how this property can be ensured by adapting the schedulers described in the previous section.

Module-Order is too Restrictive. We show in Figure 8.2 a simple stack that we use to illustrate that the module-order property is too restrictive. The stack is composed of two protocol modules: the module $ABcast_1$ implements the atomic broadcast service, while the module $RBcast_1$ implements the reliable broadcast service. Protocol modules in our stack communicate using six different event types. The events of types $Abcast$, $Rbcast$ and $Send$ are triggered to send a message respectively through the atomic broadcast, reliable broadcast and network modules. Similarly the events of types $Adeliver$, $Rdeliver$ and $Deliver$ are triggered to deliver a message from those modules.

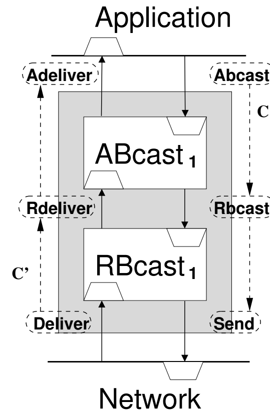

Figure 8.2: Example of computations showing the restriction of module-order.

In order to show that the module-order property is too restrictive, we consider two computations C and C' (starting in this order): computation C contains only events to send messages, while computation C' contains only events for message deliveries (see Figure 8.2). If we assume that the execution of these computations ensures the module-order property, we end up with the following execution. Because (1) module $RBcast_1$ handles an event of both C and C', and (2) C starts before C', no event of computation C' can be handled before the event of type $Rbcast$ from computation C has been handled. As a result, no events are handled concurrently in the stack.

Consider now the execution in which events of types $Deliver$ and $Abcast$, and afterwards events of types $Rdeliver$ and $Rbcast$ are handled concurrently. This execution violates the module-order property. However, because of the semantic of events types $Rbcast$ and $Deliver$, two events of these types can be handled by module $RBcast_1$ in any order without altering the properties ensured by the protocol stack. In other words, this execution is still correct despite the fact that it does not ensure the module-order property.

We have seen with this example that the module-order property does not need to be ensured for all events. We now define for which kinds of events this property has to be ensured.

Critical Event Type. An event type is *critical* if an event of that type must be handled in a specific order with respect to any other event(s) (that may be of a different type). By extension, we say that an event is critical if its type is critical. For instance, the event type $Adeliver$ in Figure 8.2 is critical, since two events of that type have to be handled in a specific order (the first event triggered must be the first to be handled). All other event types in Figure 8.2 are non-critical.

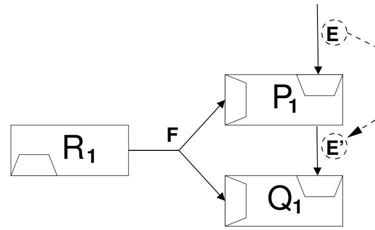

Figure 8.3: Example of critical event types.

Figure 8.3 shows another example of critical event types. The event type F denotes events that notify a change of the state of module R_1 (such as a process suspicion). Any event e' of type E' directly causally depends on an event e of type E, and has to be handled with the same knowledge of the state of R_1 as e. As a result, events of types E and E' have to be handled in a specific order with respect to events of type F. Therefore, these three event types are critical.

Path. A *path* $[e, end[$ is the set of events that causally depend on event e and do not causally depend on any event in the set end, including e, but excluding all events in the set end. A path $[e, end[$ starts when event e is triggered. Note that a computation initiated by the external event e corresponds to the path $[e, \emptyset[$.

Critical Path. A path $[e, end[$ is *critical* if and only if (1) all events of the path are critical, (2) event e does not directly causally depend on a critical event, and (3) no event in end is critical. Conditions (2) and (3) ensure that a critical path is never included in another critical path. In other words, each critical event belongs to exactly one critical path.

Based on the definition of critical paths, we are able to propose a relaxed version of the module-order property. Basically, the *relaxed module-order property* ensures the module-order property for critical paths instead of computations. As a result, the module-order property can be seen as a particular case of the relaxed module-order property where all event types are critical (and thus, critical paths correspond to computations).

Relaxed Module-Order. Consider two events e and f belonging to two different critical paths such that e belongs to a critical path that starts before the critical path that contains f. If there is a module P that handles both e and f, then e is handled before f.

Because the relaxed module-order property is a generalization of the module-order property, the scheduler for the first property is very similar to the one to ensure the latter property. Below, we describe how to adapt the simple scheduler (see Algorithm 8) to ensure the relaxed-module order property.[3]. Obviously, the two variants described in the previous section can be also adapted to the relaxed module-order property.

Adapting the Simple Module-Order Scheduler. The basic idea is to use the array of lists $modules$ to order critical paths instead of computations. This requires the following. Similarly to computations, an identifier is attached to a critical path when its starts. Furthermore, at this time, the identifier is added to the end of the list $modules[P]$, for all modules P that handle an event of the critical path. For this purpose, we assume that the set of modules that handle an event of a critical path $\mathcal{P}$ is known when $\mathcal{P}$ starts. This set can be easily computed by the scheduler if the programmer specifies (1) the critical event types, and (2) similarly to the simple module-order scheduler, the types of the events triggered by each handler. Contrary to computations, a critical path can be started upon both internal or external events. The start of a critical path can be, however, easily detected: it corresponds to the triggering of an event e such that (1) e is critical, and (2) e does not *directly* causally depend on a critical event.

Removing a critical path from the lists $modules[P]$ is similar to removing computations. For this purpose, the scheduler maintains a list of all events of the critical path that have not been executed: critical events are added to the list when they are triggered, and removed upon calls to endHandling. Once there are no more events in the list, the critical path can be removed from the lists $modules[P]$.

Two additional modifications are required. First, similarly to Algorithm 8, calls to the method startHandling(e) must block until the identifier of the critical path of e is at the head of the lists $modules[P]$, for all modules P that handle e. However, if event e is not critical, the call is non-blocking. Second, instead of returning any event of a computation id, the method next(id) returns an event e with the following property: (1) event e is not critical or (2) no event of id that has not been executed belongs to a critical path started before the critical path of e. This is necessary to prevent deadlock among computations as we now explain. Imagine two critical paths $\mathcal{P}$ and $\mathcal{P}'$ of the same computation id, such that $\mathcal{P}$ starts before $\mathcal{P}'$. Consider two events e and e' belonging respectively to critical paths $\mathcal{P}$ and $\mathcal{P}'$, that are handled by the same module. If method next(id) returns e' before e, the computation will block forever, since e' can be executed only after e.

[3] A complete description of the relaxed module-order scheduler can be found in Annex A.

8.4 Implementation

We implemented four schedulers within the SAMOA protocol framework [Sam08] (described in Section 7.5). The first three schedulers are respectively the Simple Module-Order, the Bounded Module-Order, and the Route Module-Order schedulers (described in Section 8.2). We adapted the schedulers so that the programmer can choose to ensure the module-order property or the relaxed-module order property. The last scheduler ensures sequential executions. This allowed us to compare the performance of a "concurrent" group communication stack with the performance obtained by the corresponding "sequential" stack (see Section 8.5).

Instead of being based on events, the schedulers have been adapted to service-interfaces. The adaptation is straightforward as we now explain (see Table 8.1). Every service interface corresponds to three event types; one per type of interaction (i.e., request, reply, or notification). Therefore, requests, replies and notifications are equivalent to events. Finally, each executer or listener corresponds to one handler, while each interceptor is equivalent to two handlers.

One Service-Interface	$\cong$	Three Event Types
One Reply, one Request or one Notification	$\cong$	One Event
One Executer or one Listener	$\cong$	One Handler
One Interceptor	$\cong$	Two Handlers

Table 8.1: Equivalences between service-interfaces and events.

We now present several optimizations and additional features implemented by our schedulers. For readability, we still use the event terminology.

8.4.1 Optimizations

Thread Pool. Instead of creating a new thread for each computation, our schedulers use a pool of threads. The size of the pool can be set by the programmer. When a computation C is initiated, one thread of the pool starts to execute C. If no thread is available (i.e., all threads already execute a computation), the computation is simply delayed. The use of a pool has two advantages. First, it avoids to create new threads, which is a rather costly operation. Second, it reduces the number of context switches (so their overall cost), since the number of concurrent threads is bounded.

It must be added that the scheduler adds a computation C to the list $modules[P]$ only when a thread is available to execute C (instead of adding C at the time of its initialization). This allows us to bound the size of lists $modules[P]$ by the size of the thread pool. As a result, the cost (in memory) of the operations on these lists is reduced.

Computing the Parameter dep. Remember that the parameter dep contains all the modules that may handle events of a critical path, and must be known at the initialization of a critical path (or a computation). One can observe that this parameter depends exclusively on the type of the event that initiates the critical path. Thus, it can be computed by a scheduler upon the initialization of the protocol stack (rather than on the initialization of the corresponding critical path), which minimizes the cost induced by the initialization of critical paths.

8.4.2 Additional Features

Extended Causal Order. Events of a given computation are handled with respect to the extended causal order property which has been described in [UMDK06, Men06].[4] This property provides basic ordering guarantees that are useful for protocol programmers, e.g., it ensures that two events triggered by a given handler execution are handled in the order in which they are triggered. More precisely, the property is the following. Consider two events e and f of the same computation that are triggered by the same module P. Assume that e is triggered before f. Then, each event e' that causally depends on e but does not causally depend on an event $e'' \neq e$ triggered by P after f, is handled before event f. In this case, we say that event e' causally precedes f.

Ensuring the extended causal order property requires to carefully define the set of critical event types. Indeed, the following scenario has to be avoided. Consider two critical events e and f belonging respectively to the critical paths $\mathcal{P}$ and $\mathcal{P}'$ such that $\mathcal{P}$ starts before $\mathcal{P}'$. If event f causally precedes e, we have a contradiction: (1) e must be handled before f (due to the relaxed module order property), and (2) f must be executed before e (due to the extended causal order property). However, note that we never observed such a scenario in practice although several group communication stacks (such as the ones described in [Men06]) were implemented using the SAMOA protocol framework. In any case, it is sufficient to ensure the module-order property (see definition on page 89) instead of the relaxed module-order property (see definition on page 95) to avoid such a scenario.

Atomic Tasks. In addition to triggering external events, the application can initiate *atomic tasks*. Atomic tasks allow to execute a piece of code that modifies the state of the protocol stack (e.g., to initialize or to checkpoint the protocol stack). To ensure correct executions, each atomic task T is executed atomically in the sense that no other atomic task and no computation is executed while T is executed.

Timeouts. Some modules need to periodically perform some specific tasks. For instance, consider a module P that implements a failure detector based on heartbeats [CT96]. Such a module periodically triggers an event to send a special message to other stacks, so that they can compute the set of alive stacks. To facilitate

[4]The property was slightly adapted to correspond to the context of our work, since the authors in [UMDK06, Men06] do not consider the notion of computation.

the implementation of such modules, our schedulers provide a simple interface to trigger events after a given timeout. Both the delay after which the timeout event occurs and the recurrence of the event can be set by the programmer.

To implement this special feature, our schedulers run a special thread that manages all the timeout events to be triggered. All timeout events are considered as external, and thus initiate a new computation. Finally, note that the programmer may also initiate atomic tasks after a given timeout (e.g., to checkpoint the state of a stack every t minutes).

Dynamic Bindings. Dynamically binding or unbinding handlers raises the following problem. Consider a critical path $\mathcal{P}$ that contains an event e of a type E. Assume that after the initialization of $\mathcal{P}$ (and thus, after $\mathcal{P}$ has been added to the lists $modules[P]$), but before the handling of e, a handler of module Q is bound to type E. In such an execution, the critical path $\mathcal{P}$ is not added to the list $modules[Q]$. As a result, the (relaxed) module-order property can be violated.

To solve the problem, the dynamic bindings must be performed atomically, i.e., no computation should be executed in concurrence with the dynamic bindings. Thus, one solution consists in using atomic tasks. However, such a solution requires to slightly modify the DPU algorithms described in Chapter 6. We also provide another solution. At any time, the scheduler can be turned into a sequential mode, i.e., all computations are executed sequentially. At this time, any module (such as a DPU manager) can safely dynamically bind or unbind handlers. Obviously, the scheduler can be afterwards again turned into a concurrent mode.

8.5 Performance Evaluation

The contribution of this section is twofold. First, we evaluate the amount of concurrency allowed by our schedulers. To do so, we define a simple concurrency metric. Informally, this metric measures the rate of events that are executed concurrently. Second, we evaluate the performance of a group communication middleware executed (1) with our schedulers for concurrency and (2) with our sequential scheduler. This allows us to measure the possible performance gain induced by concurrency.

In order to evaluate the amount of concurrency and the gain induced by concurrency, we use the group communication middleware described in Section 4.2.1 (see Figure 4.3). The benchmark and the performance metrics considered in this section are the ones described in Section 6.5. However, we consider here a different system setup that is composed of three two-processor machines (so that we can take advantage from concurrency). Note that the system setup considered here is less recent than the one considered for previous performance evaluations, which explains the comparatively less performant results presented below.

We start this section by describing our new system setup. We then discuss some details of our benchmark that are relevant to the present context. More specifically, we describe the critical events of the stack used in the benchmark, and the different threads that compose our benchmark. This allows the reader to understand the

different results that we present below. We next introduce our metric for concurrency, and discuss the amount of concurrency allowed by our schedulers. Finally, we discuss the performance results.

System Setup. The benchmarks were run on a cluster of three machines running Mac OS X 10.4.11 (kernel Darwin 8.11.0). Each machine has two processors PowerPC G5 (3) at 1.8 GHz and 1 GB of RAM. The machines are interconnected by 100Mbits Ethernet (which is exclusively used by the cluster machines) and run Sun's 1.5.0 Java Virtual Machine, but use a light-weight marshaling library [PHN00, HNMP05] instead of standard Java serialization [Sun04]. The machines were dedicated to the performance benchmarks and had no other load on them.

Some Remarks about the Benchmark. Figure 8.4 shows the implementation of our group communication stack from which we removed the DPU protocols (since dynamic bindings require sequential execution). We differentiate here the events to send and to deliver reliable messages through/from the network (i.e., event types $Send$ and $Deliver$) from the corresponding events for unreliable messages (i.e., event types $USend$ and $UDeliver$). We denote critical event types by dashed arrows (i.e., only the event types $Adeliver$, $Deliver$ and $Suspect$ are critical). Thus, in our stack, every critical path contains a single event (of type $Adeliver$, $Deliver$ or $Suspect$). Note that the $Deliver$ events are critical due to the following reason: these events may transport special messages to initialize the TCP connections, which have to be processed in a specific order.

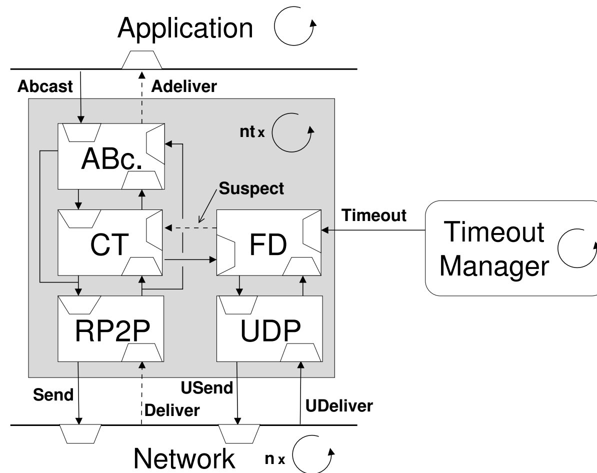

Figure 8.4: Implementation of our benchmark.

We now briefly explain how computations are executed in our stack. Computations can be initiated by four types of events: $Abcast$, $Deliver$, $UDeliver$ and $Timeout$. All modules in the stack may handle events of a computation initiated by

an event of type $Deliver$. Similarly, all modules except ABc may handle events of computations initiated by events of type $UDeliver$ or $Timeout$. Finally, only modules ABc and $RP2P$ may handle events of computations initiated by events of type $Abcast$.

The implementation of our stacks uses the following threads (which are denoted by arcs in Figure 8.4). The network uses n threads; one thread per process, including the local process. This allows to parallelize the serialization task, which is a rather costly operation (see [Men06]). Both the timeout manager and the application use a single thread. Finally, we vary the number nt of threads of the stack itself during our experiments.

Note that the implementation of the JGroups protocol stack [Ban08] has some resemblances with our implementation. Similarly to our benchmark, JGroups is based on n threads for the network layer and one thread for the application layer. However, no thread is dedicated to the execution of the protocol stack. Instead, the threads composing the network and the application layers directly execute the code implemented by the protocol stack. Moreover, the guarantees ensured upon concurrent executions of these threads in the protocol stack are unclear.

Amount of Concurrency. We define the amount of concurrency for a given execution as follows. We denote by $\mathcal{E}$ the set of events that were triggered during the execution, and by nt the number of threads executed in our stack. Let T be the time to handle all events in $\mathcal{E}$, and let T_e be the time to handle event e. Then, the amount of concurrency is defined as $\mathcal{C} = \frac{nt}{nt-1}(1 - \frac{T}{\sum_{e\in\mathcal{E}} T_e})$.

One can observe that if the execution is sequential, we have $T = \sum_{e\in\mathcal{E}} T_e$. Therefore, $\mathcal{C} = 0$. On the other hand, if concurrency is maximal, we have $T = \frac{\sum_{e\in\mathcal{E}} T_e}{nt}$, which implies that $\mathcal{C} = 1$. Informally, our concurrency metric measures the rate of events that have been executed concurrently. The purpose of the factor $\frac{nt}{nt-1}$ is to normalize the rate with respect to the maximum amount of events that can be executed concurrently.

	Module-Order	Relaxed Module-Order
Simple Scheduler	0.08	0.28
Bounded Scheduler	0.19	0.28
Route Scheduler	0.23	0.28

Table 8.2: Amount of concurrency.

Table 8.2 shows the amount of concurrency that we obtained with our three concurrent schedulers while ensuring either the module-order or the relaxed module-order property. These results were obtained in experiments with high values of the offered load l (i.e., $l \geq 1000$ msgs/s) and with 4 threads executing our stacks. Note that even if only 2 processors were available, we think that running experiments with more than 2 threads executing our stack is interesting for the following reason:

with our schedulers for concurrency, a computation (thread) may be blocked for a while (at the startHandling method, see Algorithm 7). Therefore, executing four computations concurrently may be more efficient than executing two computations concurrently, even with only two processors. We also conducted experiments with different values for these parameters (i.e., $100 \leq l < 1000$ and $1 < nt < 4$), and obtained similar values: we observe a maximum difference of 2% with the results presented in Table 8.2.

The results first show that the simple scheduler provides less concurrency while ensuring the module-order property. This can be explained by the nature of the computations in our stack. Most computations involve most of the modules in our stack, while the simple scheduler allow only concurrency between two computations that involve disjoint sets of modules. By detecting as soon as possible when a module does not handle anymore events of a computation, the bounded and the route schedulers increase the amount of concurrency (as shown in the second column of Table 8.2).

Second, as expected, we can observe that the relaxed module-order property allows more concurrency than the module-order property (compare second and third columns of Table 8.2). Contrary to the module-order property, the bounded and the route versions of our concurrency scheduler do not increase the amount of concurrency while ensuring the relaxed-module order property. This is because critical paths in our stack always contain a single event e. As a result, the bound and the route schedulers cannot detect that some module P has handled e before the end of the critical path (the time when the simple scheduler detects that the critical path can be removed from the list $modules[P]$).

Performance Results. Initially, we performed experiments using our concurrent schedulers with only *one* thread executing our stack. The results of these experiments showed that our concurrent schedulers perform as well as the sequential scheduler in such a configuration. From this observation, we can conclude that the overhead induced by our concurrent schedulers is negligible. We now discuss the results that we obtained when several threads are used by our concurrent schedulers. We only show results for the simple scheduler ensuring the relaxed module-order property, since the results with other schedulers, or ensuring the module-order property, are very similar (and lead us to the same conclusions). All experiments were performed with 64B messages.

In Figure 8.5, we show average latency (left) and throughput (right) both as a function of the offered load, when 2 (left) or 4 (right) threads execute our stack. The solid lines show the results obtained with the simple scheduler when ensuring the relaxed module-order property. To observe the influence of concurrency on the results, the graphs also include the results obtained with the sequential scheduler (see dashed lines in Figure 8.5). Obviously, the sequential scheduler uses always one single thread.

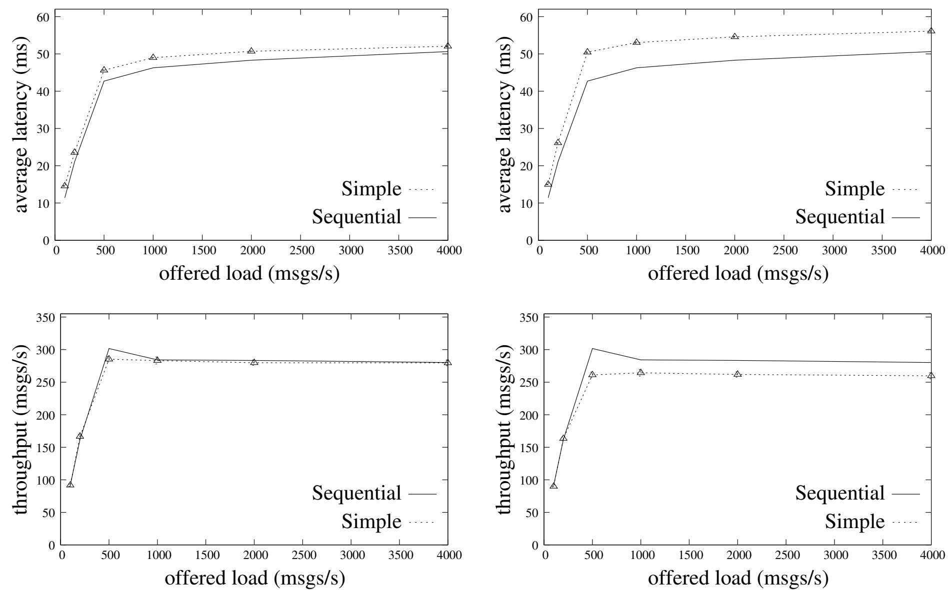

Figure 8.5: Average latency (top) and throughput (bottom) as a function of the offered load when 2 (left) or 4 (right) threads execute our stack.

Surprisingly, the results show that concurrency in the protocol stack does not improve the performance of our stack. Instead, concurrency decrease the performance of our group communication stack. This can be explained as follows. First, our benchmark is composed of several threads in addition to the threads dedicated to our schedulers (and thus, to the execution of our stack). Thus, our experiments with our sequential scheduler benefit also from the two processors of each machine. Second, all our experiments involve more threads (i.e., $nt + n + 2$, see Figure 8.4) than the available two processors. Hence, context switches occur during all our experiments. As a result, increasing the number of threads increases the number of context switches and their costs (this can be observed by comparing the results obtained with 2 and 4 threads). The gain that we can expect with concurrency is therefore ruined by the cost of context switches.

As a conclusion, our results show that concurrency to improve performance requires to carefully design the number of threads with respect to the available processors. Despite our results, we still believe that concurrency may improve performance in group communication stacks. This will require further experiments and research with machines that have more processors (or cores).

8.6 Related Work

Several algorithms to ensure correct concurrent execution in various contexts can be found in the literature. We now briefly describe the most relevant work related to ours.

Software Transactional Memory (STM). The concept of STM has been first introduced in [ST97], and has been the focus of many research over recent years (see [MIS05, CS06, HG06, HLM06, RFF06] among others). In this context, a transaction is a sequence of operations (read or write) on memory that are executed in *isolation*. The isolation property ensures that the result of executing two transactions concurrently is equivalent to *any* sequential execution of these transactions. In contrast, the module-order property ensures that the concurrent execution of two transactions (i.e., computations) is equivalent to *the* sequential execution in which the transactions are initiated in the same order as in the concurrent execution. So, the isolation property does not ensure that events are handled in a specific order, which is a necessary property in our context.

There is a second major difference between STM and our work. Most of STM implementations assume that a transaction can be aborted, and then rollbacked (i.e., all the effects of the transaction are cancelled). In our context, this is not possible, since transactions have side effects. For instance, a transaction may result in sending a message through the network. Such a message cannot be trivially cancelled. To prevent this problem (known as the output commit problem [EAWJ02]), the message must be delayed until the transaction commits (i.e., successfully terminates). However, delaying the message may significantly decrease the performance of protocol stacks.

Isolation in Database Systems. Similarly to STM, transactions in database systems are executed in isolation, and thus the above comments about STM are also relevant in the context of databases. Note that transactions in databases ensure other properties in addition to isolation (i.e., durability, atomicity and consistency). However, these properties are irrelevant in our work.

Quite a large number of concurrency control algorithms have been proposed to ensure isolation in centralized or distributed database systems. The algorithms generally fall into one of the following two basic categories: *locking* algorithms and *non-locking* algorithms. A comprehensive study of such algorithms can be found in [BHG87].

Our schedulers have some resemblance with the basic two-phase locking (the most basic locking algorithm). Similarly to the two-phase locking algorithm which takes the locks on the resources (modules) that are accessed by a transaction (in the 1st phase) and releasing the locks (in the 2nd phase), our schedulers order a transaction with respect to other transactions accessing shared resources using one list per resource (in the 1st phase), and remove the transaction from the lists (in the 2nd phase). However, the first phase in our schedulers is non-blocking, which allows two computations to access different resources in parallel, even if they access a common set of resources.

As in implementations of STM, non-locking algorithms may result in aborting transactions. Therefore, the output commit problem (and its consequences in the context of protocol stacks) can also be observed with such algorithms.

Resource Allocation. Methods for *deadlock avoidance* in allocating resources [Tan01, SGG02] are also relevant to our work, since our schedulers must ensure that each computation C (thread) can access the modules (resources) that handle an event of C. The *banker's algorithm* [Dij02] is one of the methods for deadlock avoidance that allows the highest level of concurrency. Roughly speaking, this algorithm considers each resource request as it occurs, and assigns the resource to a thread only if there is a guarantee that no deadlock can occur in the future. Similarly to our schedulers, this algorithm requires to know a priori the resources used by each thread. However, in our context, the resources can be easily computed by the scheduler itself, which is not the case with the banker's algorithm. More precisely, when a computation starts, it does not have to declare which resources will be accessed, while in the banker's algorithm, processes must explicitly provide this information.

8.7 Conclusion

We proposed in this chapter a correctness property for concurrent execution in protocol stacks and we have shown how the property can be relaxed in order to increase the level of concurrency. We presented afterwards several schedulers to transparently ensure our correctness properties, i.e., our schedulers require a minimum information from the stack programmer to ensure correct concurrent executions. All the schedulers have been implemented within our experimental protocol framework SAMOA, which allowed us to measure the amount of concurrency induced by our schedulers. The measures showed that our schedulers introduce a relatively high level of concurrency in our group communication stack.

We have also conducted some experiments to observe the gain introduced by concurrency. Unfortunately, the results that we obtained did not allow us to conclude that concurrency improves performance in the context of group communication stacks. However, we strongly believe that better results can be obtained by using machines with a sufficient number of processors or cores. The current development in the area of multiprocessor machines requires further experiments in order to validate (or invalidate) this hypothesis.

Chapter 9

Conclusion

9.1 Research Assessment

The thesis has proposed two techniques to improve the performance of group communication middleware. First, we provided solutions to dynamic protocol update, so that group communication middleware can adapt to environment changes. Second, we investigated solutions to introduce concurrency within group communication middleware in order to take advantage of multiprocessor machines. We now assess the main contributions of the thesis in more details.

Modular Approach to Dynamic Protocol Update (DPU). We have proposed a simple approach to DPU, and compared it with two of the most representative existing solutions [vRBH$^+$98, CHS01]. Our comparison showed that our solution has some advantages over these solutions. First, our solution is highly modular: it does not require to extend the updateable protocols. Second, our solution enforces DPU to be only based on the specification of the updateable protocol rather than on its implementation, which facilitates the verification of DPU. Third, our solution is highly flexible, e.g., it allows a single protocol to be replaced at a time (and does not require to replace the whole middleware in order to update a single protocol).

Predicate-Based Approach to Characterize DPU Protocols. We have introduced an elegant approach to characterize DPU protocols with a set of rules based on predicates that apply to group communication protocols. These rules concisely describe the scope of applicability of DPU protocols, i.e., they allow us to easily determine if a given protocol can be replaced by a DPU protocol. Furthermore, the rules can be easily verified independently from each other, which simplifies correctness proofs of DPU protocols.

DPU Protocols to replace Group Communication Protocols. We have described four DPU protocols that allow us to replace most protocols of a group communication middleware. More specifically, we have first presented a very simple DPU

protocol that is mostly dedicated to the replacement of low-level protocols such as reliable channels, failure detectors and reliable broadcast. Then we have designed and verified three more complex DPU protocols to replace respectively (1) consensus protocols, (2) local ordering protocols (e.g., reliable fifo), and (3) global ordering protocols (e.g., atomic broadcast). Furthermore, we have evaluated our DPU protocols for consensus and global ordering protocols. Our evaluation showed that the performance overhead induced by these DPU protocols is acceptable. This showed that our DPU protocols can be used to implement efficient adaptive group communication middleware.

Service Interface: A New Abstraction to Implement Adaptive Group Communication Middleware. We have introduced a new abstraction, called service interface, which simplifies the implementation of adaptive group communication middleware. We have then compared service interfaces with events, a well-known and widely-used abstraction to implement (adaptive) group communication middleware. The comparison highlighted the advantages of service interfaces over events: service interfaces (1) adequately model protocol interactions, (2) simplify protocol composition, and (3) provide integrated mechanisms to implement DPU. All these advantages reduce the risk of programming errors during the development of adaptive group communication middleware.

Transparent Concurrency. We have first identified a correctness property for concurrent executions within group communication middleware, and discussed how to relax the property in order to increase concurrency. We then described several schedulers that ensure our property. These schedulers hide the complexity induced by concurrency from the programmer (who has the impression that there is no concurrency). Hence, introducing concurrency in a group communication middleware is greatly simplified.

We also showed that our schedulers are efficient in the sense that they introduce a relatively high level of concurrency. Unfortunately, we could not observe a gain in performance resulting from the gain in concurrency. We believe that this is mainly due to the few number of processors on the machines used for our experiments, and strongly think that better results will be obtained with machines with additional processors.

The SAMOA Protocol Framework. We have implemented all our ideas within SAMOA, our service-interface-based protocol framework dedicated to the development of concurrent and adaptive group communication middleware. SAMOA is light-weight (3600 lines of Java code), and exhibits good performance with respect to Cactus [Cac01, BHSC98], an efficient event-based framework. To validate our framework, SAMOA was used to implement a concurrent and adaptive group communication middleware based on (1) the Fortika toolkit [MRS06, Men06] and (2) the DPU protocols that have been presented in the thesis.

9.2 Open Questions and Future Research Directions

DPU Protocols for Other System Models. All DPU protocols that have been presented in the thesis assume a crash-stop model with static groups. However, we believe that these DPU protocols can be rather easily adapted to different system models. For instance, in our opinion, our DPU protocol to replace global ordering protocols requires only slight modification in order to correctly replace atomic broadcast protocols designed for the crash-recovery model with static groups (e.g., [MS05]). The main modification is to periodically save the structure of protocol stacks (i.e., module bindings and protocols composing the stack), so that crashed stacks can recover with a consistent state.

State Transfer. We discussed in Section 3.5 the problem of transferring some state from the protocol that gets replaced to the new protocol when the replacement occurs. Our approach to DPU currently does not consider this problem, but we believe that it can be extended to allow state transfer. This raises several issues: for instance, it should be clearly defined which part of the protocol state must be transferred (e.g., the state that is particular to a given implementation (protocol) should not be transferred). We are considering to extend our approach with state transfer as a future work.

Dynamic Addition/Removal of DPU Protocols. We showed in Chapter 6 that our DPU protocols induced some performance overhead when no replacement occurs. We think that this cost can be suppressed with the following approach. Initially, the group communication middleware does not contain any DPU protocols. A DPU protocol is dynamically added to the middleware only when a replacement is required, and dynamically removed immediately after the replacement terminates. We planned to extend our approach to DPU in order to allow dynamic addition and removal of DPU protocols.

Efficient Self-Adaptive Group Communication Middleware. We discussed in Section 3.4 some solutions, based on DPU, to render a group communication middleware self-adaptive. However, to our knowledge, none of these solutions propose efficient strategies for self-adaptation. For instance, it is not clear which environment changes can be expected in practice, and which protocols are optimal for a given environment. Therefore, we think that an interesting continuation of our work is to design efficient self-adaptation strategies, and implement them within our adaptive group communication middleware.

Transparent Concurrency and Further Experimental Research. As mentioned in Chapter 8, additional experiments with powerful multiprocessor machines (with more than 2 processors or cores) are required in order to show that concurrency improves performance of group communication middleware. Apart from this, we

believe that some additional issues can be explored to improve concurrency within group communication middleware. For instance, it is not clear yet if one thread per computation (which is the case of our implementation) is a better strategy than one thread per protocol. In addition, we think that our execution model can be slightly relaxed to improve concurrency, e.g., by allowing several events of the same computation to be executed in parallel.

Bibliography

[AC03] M. Aksit and Z. Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *ICDCS Workshops*, pages 84–89. IEEE Computer Society, 2003.

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *DISC*, volume 1499 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 1998.

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[App01] Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa. *The Appia project*, 2001. Documentation available electronically at `http://appia.di.fc.ul.pt/`.

[AV90] J. L. Armstrong and S. R. Virding. Erlang – An experimental telephony switching language. In *Proc. XIII International Switching Symposium*, pages 655–670, 1990.

[Ban08] B. Ban. *The JGroups Project*, 2008. Documentation available electronically at `http://www.jgroups.org/javagroupsnew/docs/index.html`.

[BBI+00] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Middleware*, volume 1795 of *Lecture Notes in Computer Science*, pages 164–184. Springer, 2000.

[BCHS01] P. G. Bridges, W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Supporting coordinated adaption in networked systems. In *HotOS*, page 162. IEEE Computer Society, 2001.

[BCRP98] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998.

[BD93] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

[BDM01] Ö. Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, 2001.

[BGT+01] F. V. Brasileiro, F. Greve, F. Tronel, M. Hurfin, and J.-P. L. Narzul. Eva: An event-based framework for developing specialized communication protocols. In *NCA*, pages 108–121. IEEE Computer Society, 2001.

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

[BHSC98] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, 1998.

[Bir93] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, 103, 1993.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[BKvRC01] M. Bickford, C. Kreitz, R. van Renesse, and R. L. Constable. An experiment in formal design using meta-properties. In *Proc. DISCEX-II '01: the 2nd DARPA Information Survivability Conference and Exposition*, pages 100–107. IEEE, June 2001.

[BKvRL01] M. Bickford, C. Kreitz, R. van Renesse, and X. Liu. Proving hybrid protocols correct. In *TPHOLs*, volume 2152 of *Lecture Notes in Computer Science*, pages 105–120. Springer, 2001.

[BMN05] D. C. Bünzli, S. Mena, and U. Nestmann. Protocol composition frameworks a header-driven model. In *NCA*, pages 243–246. IEEE Computer Society, 2005.

[BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems (2nd Ed.), Chapter 8*, pages 199–216, 1993.

[BNS+05] J. Balasubramanian, B. Natarajan, D. C. Schmidt, A. S. Gokhale, J. Parsons, and G. Deng. Middleware support for dynamic component updating. In *OTM Conferences (2)*, volume 3761 of *Lecture Notes in Computer Science*, pages 978–996. Springer, 2005.

[BO83] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC*, pages 27–30. ACM, 1983.

[Cac01] The University of Arizona, Computer Science Department. *The Cactus project*, 2001. Documentation available electronically at `http://www.cs.arizona.edu/cactus/`.

[CB03] B. Charron-Bost. Comparing the atomic commitment and consensus problems. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 29–34. Springer, 2003.

[CBCP02] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[CBG+08] G. Coulson, G. S. Blair, P. Grace, F. Taïani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1):1:1–1:42, 2008.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[CHD98] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *PODC*, pages 237–246. ACM, 1998.

[CHS01] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *ICDCS*, pages 635–643. IEEE Computer Society, 2001.

[CHTCB96] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *PODC*, pages 322–330. ACM, 1996.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

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

[CS06] J. P. Cachopo and A. R. Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

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

[DDS87] D. Dolev, C. Dwork, and L. J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of ACM*, 34(1):77–97, 1987.

[Dij02] E. W. Dijkstra. Cooperating sequential processes. *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 65–138, 2002.

[DL02] P.-C. David and T. Ledoux. An infrastructure for adaptable middleware. In *CoopIS/DOA/ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 773–790. Springer, 2002.

[DLS88] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[DSU04] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[Dug01] D. Duggan. Type-based hot swapping of running modules. In *ICFP*, pages 62–73. ACM, 2001.

[EAWJ02] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[EMPS04] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards flexible finite-state-machine-based protocol composition. In *NCA*, pages 281–286. IEEE Computer Society, 2004.

[EMS95] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *ICDCS*, pages 296–306. IEEE Computer Society, 1995.

[Ens01] Department of Computer Science, Cornell University. *The Ensemble project*, 2001. Documentation available electronically at `http://dsl.cs.technion.ac.il/projects/Ensemble/`.

[ES05] R. Ekwall and A. Schiper. Replication: Understanding the advantage of atomic broadcast over quorum systems. *Journal of Universal Computer Science*, 11(5):703–711, 2005.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *FCT*, volume 158 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 1983.

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

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, pages 150–162. ACM, 1979.

[Gra86] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[Hay98] M. Hayden. The ensemble system. Technical report, Computer Science Department, Cornell University, Ithaca, NY, USA, 1998.

[HG06] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Memory System Performance and Correctness*, pages 82–91. ACM, 2006.

[HHD98] R. Hayton, A. Herbert, and D. I. Donaldson. Flexinet - a flexible component oriented middleware system. In *ACM SIGOPS European Workshop*, pages 17–24. ACM, 1998.

[Hic01] M. Hicks. *Dynamic Software Updating*. PhD thesis, Computer and Information Science Department, University of Pennsylvania, August 2001.

[HLA03] J. Hallstrom, W. Leal, and A. Arora. Scalable evolution of highly available systems. *Transactions of the Institute for Electronics, Information and Communication Engineers (IEICE)*, E86-B(10):2154–2166, 2003.

[HLM06] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262. ACM, 2006.

[HNMP05] B. Haumacher, C. Nester, T. Moschny, and M. Philippsen. *JavaParty: Fast Object Serialization with `uka.transport`*. Universität Karlsruhe, Fakultät für Informatik, Germany, 2005. Documentation available electronically at `http://svn.ipd.uni-karlsruhe.de/trac/javaparty/wiki/uka.transport`.

[HS07] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *DSN*, pages 92–101. IEEE Computer Society, 2007.

[HT94] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[KG07] S. Karmakar and A. Gupta. Adaptive broadcast by distributed protocol switching. In *SAC*, pages 588–589. ACM, 2007.

[KRL+00] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. Monitoring, security, and dynamic configuration with the *dynamic*TAO reflective ORB. In *Middleware*, volume 1795 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2000.

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

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

[Lap91] J.-C. Laprie. *Dependability: Basic concepts and terminology in English, French, German, Italian, and Japanese (volume 5 of Dependable computing and fault-tolerant systems)*. Springer-Verlag, 1991.

[LC05] Y.-F. Lee and R.-C. Chang. Developing dynamic-reconfigurable communication protocol stacks using Java. *Software Practice & Experience*, 35(6):601–620, 2005.

[Led99] T. Ledoux. OpenCorba: A Reflective Open Broker. In *Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 1999.

[LvRB+01] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting meta-properties. In *ICDCS Workshops*, pages 37–42. IEEE Computer Society, 2001.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mal96] C. P. Malloth. *Conception and implementation of a toolkit for building fault-tolerant distributed applications in large scale networks*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1996. Number 1557.

[MCGS03] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. cactus: Comparing protocol composition frameworks. In *SRDS*, pages 189–198. IEEE Computer Society, 2003.

[Men06] S. Mena. *Protocol Composition Frameworks and Modular Group Communication: Models, Algorithms and Architectures*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2006. Number 3633.

[Mic01] Microsoft. *COM+*, 2001. Documentation available electronically at `http://www.microsoft.com/com/default.mspx`.

[MIS05] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2005.

[MKS89] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, 1989.

[MPG+00] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP*, volume 1850 of *Lecture Notes in Computer Science*, pages 337–361. Springer, 2000.

[MPR01] H. Miranda, A. S. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *ICDCS*, pages 707–710. IEEE Computer Society, 2001.

[MR99] A. Mostéfaoui and M. Raynal. Solving consensus using chandratoueg's unreliable failure detectors: A general quorum-based approach. In *DISC*, volume 1693 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 1999.

[MR06] J. Mocito and L. Rodrigues. Run-time switching between total order algorithms. In *Euro-Par*, volume 4128 of *Lecture Notes in Computer Science*, pages 582–591. Springer, 2006.

[MRA+05] J. Mocito, L. Rosa, N. Almeida, H. Miranda, L. Rodrigues, and A. Lopes. Context adaptation of the communication stack. In *ICDCS Workshops*, pages 652–655. IEEE Computer Society, 2005.

[MRS06] S. Mena, O. Rütti, and A. Schiper. *Fortika: Robust Group Communication*. EPFL, Laboratoire de Systèmes Répartis, may 2006. Documentation available electronically at `http://lsrwww.epfl.ch/fortika/`.

[MS05] S. Mena and A. Schiper. A new look at atomic broadcast in the asynchronous crash-recovery model. In *SRDS*, pages 202–214. IEEE Computer Society, 2005.

[MSKC04] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[MSW03] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 414–432. Springer, 2003.

[MY98] S. Mishra and R. Yang. Thread-based vs event-based implementation of a group communication service. In *IPPS/SPDP*, pages 398–402. IEEE Computer Society, 1998.

[Nek01] Ecole Polytechnique Fédérale de Lausanne, Distributed Systems Laboratory. *The Neko project*, 2001. Documentation available electronically at `http://lsrwww.epfl.ch/neko/`.

[NLLY03] D. M. Nicol, J. Liu, M. Liljenstam, and G. Yan. Simulation of large scale networks I: simulation of large-scale networks using SSF. In *Winter Simulation Conference*, pages 650–657. ACM, 2003.

[OMG04] Object Management Group. *CORBA Component Model*, 2004. Documentation available electronically at `http://www.omg.org/`.

[PGK88] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD Conference*, pages 109–116. ACM, 1988.

[PHN00] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency - Practice and Experience*, 12(7):495–518, 2000.

[PRS08] S. Pleisch, O. Rütti, and A. Schiper. On the Specification of Partitionable Group Membership. In *EDCC*. IEEE Computer Society, 2008.

[PS02] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[RBH+98] O. Rodeh, K. P. Birman, M. Hayden, Z. Xiao, and D. Dolev. The architecture and performance of security protocols in the Ensemble group communication system. Technical Report TR-98-1703, Computer Science Department, Cornell University, 1998.

[RFF06] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2006.

[RMES07] O. Rütti, S. Mena, R. Ekwall, and A. Schiper. On the cost of modularity in atomic broadcast. In *DSN*, pages 635–644. IEEE Computer Society, 2007.

[RW04] N. D. Ryan and A. L. Wolf. Using event-based translation to support dynamic protocol evolution. In *ICSE*, pages 408–417. IEEE Computer Society, 2004.

[Sam08] Ecole Polytechnique Fédérale de Lausanne, Distributed Systems Laboratory. *The SAMOA project*, 2008. Documentation available electronically at `http://lsrwww.epfl.ch/samoa/`.

[Sch93] F. B. Schneider. Replication management using the state-machine approach. *Distributed systems (2nd Ed.), Chapter 7*, pages 169–197, 1993.

[Sch06] A. Schiper. Dynamic group communication. *ACM Distributed Computing*, 18(5):359–374, 2006.

[SDL00] The SDL Forum Society. *The SDL project*, 2000. Documentation available electronically at `http://www.sdl-forum.org/SDL/`.

[SGG02] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts, Sixth Edition*. John Wiley, 2002.

[SHB+05] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *POPL*, pages 183–194. ACM, 2005.

[SPW03] N. Sridhar, S. M. Pike, and B. W. Weide. Dynamic module replacement in distributed protocols. In *ICDCS*, pages 620–627. IEEE Computer Society, 2003.

[ST97] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[ST06] A. Schiper and S. Toueg. From set membership to group membership: A separation of concerns. *IEEE Transactions on Dependable and Secure Computing*, 3(1):2–12, 2006.

[Sun04] Sun Microsystems. *Java Object Serialization Specification*, 2004. Documentation available electronically at `http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html`.

[Sun06] Sun Microsystems. *Entreprise JavaBeans*, 2006. Documentation available electronically at `http://java.sun.com/products/ejb/`.

[Tan01] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

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

[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, 2002.

[UMDK06] P. Urbán, S. Mena, X. Défago, and T. Katayama. Concurrency in microprotocol frameworks. Research Report IS-RR-2006-004, Japan Advanced Institute of Science and Technology, Ishikawa, Japan, February 2006.

[Urb03] P. Urbán. *Evaluating the Performance of Distributed Agreement Algorithms: Tools, Methodology and Case Studies*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2003. Number 2824.

[vRBH+98] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. A. Karr. Building adaptive systems using ensemble. *Software, Practice & Experience*, 28(9):963–979, 1998.

[vRBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[WKG00] C. Walton, D. Kirli, and S. Gilmore. An abstract machine model of dynamic module replacement. *Future Generation of Computer Systems*, 16(7):793–808, 2000.

[WMK94] B. Whetten, T. Montgomery, and S. M. Kaplan. A high performance totally ordered multicast protocol. In *Dagstuhl Seminar on Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 33–57. Springer, 1994.

[Zad72] L. A. Zadeh. A rationale for fuzzy control. *Dynamic Systems*, 94(Series G):3–4, 1972.

Appendix A

Scheduler for the Relaxed Module-Order Property

Algorithm 9 describes in details our simple relaxed module-order scheduler (that is briefly presented in Section 8.3). It assumes that, upon triggering of an event e (internal or external), the set of modules that may handle an event that belongs to the critical path initiated by e is passed as a parameter of the methods internal-Trigger and externalTrigger (see parameter dep in lines 7 and 11). If event e does not initiate a critical path, we assume that the set dep is empty. Similarly to the simple module-order scheduler, this set can be easily computed by the scheduler if, for each handler h, the programmer specifies the type of the events that may be triggered by h. In addition, the programmer has to specify the critical event types.

Similarly to computations, a critical path is identified by an integer. We denote by $\mathcal{P}(e)$, the critical path to which event e belongs. If event e is not critical (and thus, does not belong to a critical path) and only in this case, $\mathcal{P}(e)$ is equal to zero.

Similarly to the simple module-order scheduler, Algorithm 9 maintains the arrays of lists $modules$ and $computations$. However, the list $modules[P]$ orders critical path instead of computations. In addition, Algorithm 9 maintains an additional array of lists, called $criticalPaths$, which contain for each critical path the events that have not been executed. We now describe how the scheduler ensures the module-order property by describing the following procedures:

- **externalTrigger(**e, dep**)** first performs the task related to the triggering of an event by calling internalTrigger at line 8 (see below). Then, the new task that executes the computation of e is initiated (line 9), and the identifier $nextId$ is incremented to correspond to the next computation (line 10).
- **internalTrigger(**e, dep**)** first adds the event e to the list of events of the computation (line 12). Then, if the event initiates a critical path (i.e., $dep \neq nil$), the variable $nextPId$ (1) is added to the lists $modules[P]$, such that $P \in dep$, and (2) is incremented to correspond to the next critical path (lines 13-16). Finally, if the event is critical (i.e., $\mathcal{P}(e) \neq 0$), event e is added to the list $criticalPath[P(e)]$ (lines 17-18).

- **next(**id**)** returns an event e of the computation identified by id, such that there is no event e' of computation id such that $\mathcal{P}(e') < \mathcal{P}(e)$ and e' has not been executed (lines 19-21).
- **startHandling(**e**)** blocks until $\mathcal{P}(e)$ is at the head of all lists of modules that handle e if and only if e is critical (lines 22-25).
- **endHandling(**e**)** first updates the list $computations[C(e)]$ at line 27. If event e is critical, it is removed from the list corresponding to its critical path (line 29). Finally, when this list is empty (i.e., the critical path terminates), the identifier $\mathcal{P}(e)$ is removed from all lists $modules[P]$ that contain $\mathcal{P}(e)$ (lines 30-33).

Algorithm 9 Simple Scheduler for the Relaxed Module-Order Property

```
1: Initialisation:
2:    modules[] ← [λ, ....]      {The lists of critical paths that may execute a given protocol module;
                                  one list per protocol module; all lists are initially empty (λ)}
3:    computations[] ← [λ, ...]  {The lists of events that are triggered but not handled;
                                  one list per computation; all lists are initially empty (λ)}
4:    nextCId ← 1                {The identifier for the next computation to be initiated}
5:    criticalPaths[] ← [λ, ...] {The lists of events that are triggered but not handled;
                                  one list per critical path; all lists are initially empty (λ)}
6:    nextPId ← 1                {The identifier for the next critical path to be initiated}

7: procedure externalTrigger(e, dep)
8:    internalTrigger(e, dep)
9:    new task computation(nextCId, e)
10:   nextCId ← nextCId + 1

11: procedure internalTrigger(e, dep)
12:   add e to the end of list computations[C(e)]
13:   if dep ≠ ∅ then                                 {Event e starts a critical path}
14:      for all P ∈ dep do
15:         add nextPId to the end of the list modules[P]
16:      nextPId ← nextPId + 1
17:   if P(e) ≠ 0 then                                {Event e is part of a critical path}
18:      add e to the end of list criticalPaths[P(e)]

19: procedure next(id)
20:   pathId ← minimal value of P(e) such that e ∈ computations[id]
21:   return any event e ∈ computations[id] such that P(e) = pathId  (nil if computations[id] = λ)

22: procedure startHandling(e)
23:   if P(e) ≠ 0 then                                {Event e is part of a critical path}
24:      for all modules P such that P handles event e do
25:         wait until P(e) is at the head of list modules[P]

26: procedure endHandling(e)
27:   remove e from the list computations[C(e)]
28:   if P(e) ≠ 0 then                                {Event e is part of a critical path}
29:      remove e from list criticalPaths[P(e)]
30:      if criticalPaths[P(e)] = λ then              {The critical path of e terminated}
31:         for all modules P do
32:            if P(e) ∈ modules[P] then
33:               remove P(e) from list modules[P]
```

List of publications

Published Parts of this Thesis

Chapters 4 and 6

[RWS05] Olivier Rütti, Paweł T. Wojciechowski, and André Schiper. Dynamic update of distributed agreement protocols. Technical Report IC-2005-012, I&C, EPFL, 2005.

[RWS06b] Olivier Rütti, Pawel T. Wojciechowski, and André Schiper. Structural and Algorithmic Issues of Dynamic Protocol Update. In *IPDPS*. IEEE Computer Society, 2006.

Chapters 5 and 6

[RS08] Olivier Rütti and André Schiper. A Predicate-Based Approach to Dynamic Protocol Update in Group Communication. In *IPDPS*. IEEE Computer Society, 2008.

Chapter 7

[RWS06a] Olivier Rütti, Pawel T. Wojciechowski, and André Schiper. Service interface: a new abstraction for implementing and composing protocols. In *SAC*, pages 691–696. ACM, 2006.

Chapter 8

[WRS04] Pawel T. Wojciechowski, Olivier Rütti, and André Schiper. Samoa: Framework for synchronisation augmented microprotocol approach. In *IPDPS*. IEEE Computer Society, 2004.

Other Publications in the Context of this Thesis

[WR05] Pawel T. Wojciechowski and Olivier Rütti. On correctness of dynamic protocol update. In *FMOODS*, volume 3535 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2005.

[BFM+06] Daniel C. Bünzli, Rachele Fuzzati, Sergio Mena, Uwe Nestmann, Olivier Rütti, André Schiper, and Pawel T. Wojciechowski. Advances in the design and implementation of group communication middleware. In *Research Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*, pages 172–194. Springer, 2006.

[RMES07] Olivier Rütti, Sergio Mena, Richard Ekwall, and André Schiper. On the cost of modularity in atomic broadcast. In *DSN*, pages 635–644. IEEE Computer Society, 2007.

[PRS08] Stefan Pleisch, Olivier Rütti, and André Schiper. On the Specification of Partitionable Group Membership. In *EDCC*. IEEE Computer Society, 2008.

Curriculum Vitæ

I was born in Neuchâtel (Switzerland) in October 1980. I attended primary and secondary school in Marin-Epagnier and Saint-Blaise. In 1998, I obtained a *Maturité Fédérale Scientifique* (Swiss baccalaureat of sciences) from the *Gymnase Cantonal de Neuchâtel*, and was awarded the *Prix Alcatel Câble SA* for the best average grade in Physics. Then I started studying at *École Polytechnique Fédérale de Lausanne* (EPFL, Switzerland), and obtained in 2003 a M.Sc. degree in Computer Science.

Since June 2003, I have been working at the Distributed Systems Laboratory (LSR) at EPFL as a research and teaching assistant, and as a PhD student under the guidance of Professor André Schiper. During 2005-2006, I was also a system administrator at LSR.