PROTOCOL COMPOSITION FRAMEWORKS AND MODULAR GROUP COMMUNICATION: MODELS, ALGORITHMS AND ARCHITECTURES

THÈSE N^{O} 3633 (2006)

PRÉSENTÉE LE 24 NOVEMBRE 2006

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Laboratoire de systèmes répartis

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Sergio MENA DE LA CRUZ

ingénieur en informatique, Universidad Politécnica, Valencia, Espagne
de nationalité espagnole

acceptée sur proposition du jury:

Prof. M. Odersky, président du jury
Prof. A. Schiper, directeur de thèse
Prof. R. Jiménez-Peris, rapporteur
Prof. D. Kostic, rapporteur
Prof. R. Schlichting, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2006

Abstract

It is noticeable that our society is increasingly relying on computer systems. Nowadays, computer networks can be found at places where it would have been unthinkable a few decades ago, supporting in some cases critical applications on which human lives may depend. Although this growing reliance on networked systems is generally perceived as technological progress, one should bear in mind that such systems are constantly growing in size and complexity, to such an extent that assuring their correct operation is sometimes a challenging task. Hence, dependability of distributed systems has become a crucial issue, and is responsible for an important body of research over the last years.

No matter how much effort we put on ensuring our distributed system's correctness, we will be unable to prevent crashes. Therefore, designing distributed systems to *tolerate* rather than *prevent* such crashes is a reasonable approach. This is the purpose of fault-tolerance. Among all techniques that provide fault tolerance, replication is the only one that allows the system to mask process crashes. The intuition behind replication is simple: instead of having one instance of a service, we run several of them. If one of the replicas crashes, the rest can take over so that the crash does not prevent the system from delivering the expected service. A replicated service needs to keep all its replicas consistent, and group communication protocols provide abstractions to preserve such consistency.

Group communication toolkits have been present since the late 80s. At the beginning, they were monolithic and later on they became modular. Modular group communication toolkits are composed of a set of off-the-shelf protocol modules that can be tailored to the application's needs. Composing protocols requires to set up basic rules that define how modules are composed and interact. Sometimes, these rules are devised exclusively for a particular protocol suite, but it is more sensible to agree on a carefully chosen set of rules and reuse them: this is the essence of protocol composition frameworks.

There is a great diversity of protocol composition frameworks at present, and none is commonly considered the best. Furthermore, any attempt to defend a framework as being the best finds strong opposition with plenty of arguments pointing out its drawbacks.

Given the complexity of current group communication toolkits and their configurability requirements, we believe that research on modular group communication and protocol composition frameworks must go hand-in-hand. The main

goal of this thesis is to advance the state of the art in these two fields jointly and demonstrate how protocols can benefit from frameworks, as well as frameworks can benefit from protocols.

The thesis is structured in three parts. Part I focuses on issues related to protocol composition frameworks. Part II is devoted to modular group communication. Finally, Part III presents our modular group communication prototype: *Fortika*. Part III combines the results of the two previous parts, thereby acting as the convergence point.

At the beginning of Part I, we propose four perspectives to describe and compare frameworks on which we base our research on protocol frameworks. These perspectives are: composition model (how the composition looks like), interaction model (how the components interact), concurrency model (how concurrency is managed within the framework), and interaction with the environment (how the framework communicates with the outside world).

We compare Appia and Cactus, two relevant protocol composition frameworks with a very different design. Overall, we cannot tell which framework is better. However, a thorough comparison using the four perspectives mentioned above showed that Appia is better in certain aspects, while Cactus is better in other aspects.

Concurrency control to avoid race conditions and deadlocks should be ensured by the protocol framework. However this is not always the case. We survey the concurrency model of eight protocol composition frameworks and propose new features to improve concurrency management.

Events are the basic mechanism that protocol modules use to communicate with each other. Most protocol composition frameworks include events at the core of their interaction model. However, events are seemingly not as good as one may expect. We point out the drawbacks of events and propose an alternative interaction scheme that uses message headers instead of events: the header-driven model.

Part II starts by discussing common features of traditional group communication toolkits and the problems they entail. Then, a new modular group communication architecture is presented. It is less complex, more powerful, and more responsive to failures than traditional architectures.

Crash-recovery is a model where crashed processes can be restarted and continue where they were executing just before they crashed. This requires to log the state to disk periodically. We argue that current specifications of atomic broadcast (an important group communication primitive) are not satisfactory. We propose a novel specification that intends to overcome the problems we spotted in existing specifications. Additionally, we come up with two implementations of our atomic broadcast specification and compare their performance.

Fortika is the main prototype of the thesis, and the subject of Part III. Fortika is a group communication toolkit written in Java that can use third-party frameworks like Cactus or Appia for composition. Fortika was the testbed for architectures, models and algorithms proposed in the thesis.

Finally, we performed software-based fault injection on Fortika to assess its

fault-tolerance. The results were valuable to improve the design of Fortika.

Keywords: distributed systems, group communication, consensus, group membership, atomic broadcast, failure detectors, fault tolerance, crash-recovery, generic broadcast, protocol architecture, header-driven, concurrency, protocol composition, frameworks, microprotocols, modular, events, fault injection, Fortika, Java, Appia, Cactus, Samoa

Résumé

Chacun sait que notre société dépend de plus en plus des systèmes informatiques. De nos jours, on peut trouver des réseaux informatiques dans des endroits où cela aurait été impensable quelques décennies auparavant, ces réseaux offrant dans certains cas une infrastructure critique puisque des vies humaines peuvent en dépendre. Bien que ce recours grandissant aux systèmes en réseau soit généralement perçu comme un progrès technique, il faudrait toutefois être conscient que de tels systèmes voient leur taille et leur complexité augmenter constamment, à tel point que le fait d'assurer leur fonctionnement correct est parfois une tâche exigeante. En conséquence, la sûreté de fonctionnement des systèmes distribués est devenue une question capitale, qui a suscité un nombre important de recherches ces dernières années.

Quel que soit l'effort que nous déployons à assurer la correction de notre système distribué, nous serons incapables d'empêcher des défaillances. Par conséquent, concevoir notre système distribué de manière à ce qu'il *tolère* des pannes plutôt qu'il les *empêche* est une approche raisonnable. C'est le but de la tolérance aux pannes. Parmi toutes les techniques qui fournissent de la tolérance aux pannes, la réplication est la seule qui permette au système de masquer les défaillances. L'intuition derrière le concept de réplication est simple : au lieu d'avoir une seule instance d'un service, on en exécute plusieurs. Si une des répliques tombe en panne, celles qui restent peuvent reprendre le travail de manière à ce que la panne n'empêche pas le système de fournir le service attendu. Un service répliqué a besoin de maintenir toutes ses répliques dans un état cohérent, et les protocoles de communication de groupe fournissent des abstractions permettant de maintenir une telle cohérence.

Les *outils de communication de groupe* existent depuis la fin des années quatre-vingts. Au début, ils étaient monolithiques et plus tard ils sont devenus modulaires. Les outils de communication modulaires sont composés d'un ensemble de protocoles déjà prêts qui peuvent être combinés selon les besoins de l'application. La composition de protocoles requiert de mettre en place des règles définissant la manière dont les modules sont composés et interagissent. Parfois, ces règles sont conçues uniquement pour quelques protocoles particuliers, mais il est plus sensé de se mettre d'accord sur un ensemble de règles soigneusement choisies et de les réutiliser : ceci est l'essence des *cadriciels (frameworks) de composition de protocoles*.

De nos jours, il y a une grande diversité de cadriciels de composition de protocoles, et aucun ne fait l'unanimité. De plus, n'importe quelle tentative de présenter un cadriciel comme étant le meilleur rencontre immédiatement une forte opposition avec un grand nombre d'arguments démontrant ses points faibles.

Étant donné la complexité des outils de communication de groupe et leurs besoins de configurabilité, nous estimons que la recherche dans le domaine de la communication de groupe modulaire et des cadriciels de composition de protocoles doit être menée en parallèle. L'objectif principal de cette thèse est de faire avancer l'état de l'art conjointement dans ces deux domaines et de démontrer à quel point les protocoles peuvent améliorer la conception des cadriciels, en même temps que les cadriciels améliorent la conception des protocoles.

La thèse est structurée en trois parties. La partie I met l'accent sur des questions relatives aux cadriciels de composition de protocoles. La partie II est consacrée à la communication de groupe modulaire. Finalement, la partie III présente notre prototype de communication de groupe modulaire : *Fortika*. La partie III combine les résultats des deux parties précédentes, constituant ainsi leur point de convergence.

Au début de la partie I, nous proposons quatre perspectives pour décrire et comparer les cadriciels sur lesquels nous basons notre recherche en cadriciels de composition de protocoles. Ces perspectives sont : le modèle de composition (à quoi ressemble la composition), le modèle d'interaction (comment les composants interagissent entre eux), le modèle de concurrence (comment la concurrence est gérée à l'intérieur du cadriciel), et l'interaction avec l'environnement (comment le cadriciel communique avec le monde extérieur).

Nous comparons Appia et Cactus, deux cadriciels de composition de protocoles significatifs, mais très différents l'un de l'autre. De manière générale, nous ne pouvons pas dire lequel est le meilleur : une comparaison minutieuse utilisant les quatre perspectives mentionnées plus haut a montré qu'Appia est meilleur pour certains aspects, tandis que Cactus est meilleur pour d'autres aspects.

Les cadriciels de composition de protocoles devraient assurer un contrôle de la concurrence afin d'éviter des *conflits d'accès* et des situations d'interbloquage. Malheureusement, ce n'est pas toujours le cas. Nous examinons le modèle de concurrence des huit cadriciels de composition de protocoles et nous proposons de nouvelles idées pour améliorer la gestion de la concurrence.

Les *événements* sont le mécanisme de base que les protocoles utilisent pour communiquer entre eux. La plupart des cadriciels de composition de protocoles utilisent des événements au coeur de leur modèle d'interaction. Cependant, les événements ne sont apparemment pas aussi bons que l'on pourrait s'y attendre. Nous mentionnons les inconvénients des événements et nous proposons un nouveau modèle d'interaction qui utilise les en-têtes des messages à la place d'événements : le modèle "dirigé par les en-têtes".

La partie II commence par un examen des caractéristiques communes aux outils traditionnels de communication de groupe et des problèmes qu'ils occasionnent. Une nouvelle architecture de communication de groupe modulaire est ensuite présentée. Cette dernière est moins complexe, plus puissante et réagit mieux

aux défaillances que les architectures traditionnelles.

Crash-recovery est un modèle où les processus tombés en panne peuvent être redémarrés et continuer leur exécution à l'endroit où ils étaient juste avant qu'ils ne tombent en panne. L'idée consiste à enregistrer périodiquement leur état sur disque. Nous défendons le point de vue que les spécifications de diffusion atomique (une primitive de communication de groupe importante) actuelles ne sont pas satisfaisantes. Nous proposons une nouvelle spécification visant à corriger les problèmes des spécifications existantes. De plus, nous avons réalisé deux implémentations de notre spécification de diffusion atomique, et nous comparons leur performance.

Fortika est le principal prototype de cette thèse, ainsi que le sujet de la partie III. Fortika est un outil de communication de groupe écrit en Java et pouvant utiliser des plates-formes comme Cactus ou Appia. Fortika constitue le banc d'essai des architectures, des modèles et des algorithmes proposés dans cette thèse.

Finalement, nous avons soumis Fortika à des injections de fautes afin d'évaluer sa robustesse. Les résultats ont contribué à améliorer Fortika de manière significative.

Mots-clés : systèmes distribués, communication de groupe, consensus, diffusion atomique, appartenance à groupes, détecteurs de pannes, tolérance aux fautes, diffusion générique, architecture de protocoles, orienté à en-têtes, concurrence, composition de protocoles, cadriciels, schémas, microprotocoles, modulaire, événements, injection de fautes, Fortika, Java, Appia, Cactus, Samoa

In Memoriam

Dionisio Mena Gil

Acknowledgments

When I first arrived in Lausanne, I did not know what a *fondue* or a *raclette* was, I thought there were plenty of holes in a Gruyere cheese, I had never been higher than 2000 meters of altitude and, as the last straw, I did not even look at the others' eyes when drinking a toast! Today, almost six years later, when people speak of Switzerland I feel something very special, I feel they are talking about a wonderful period of my life, a period with plenty of enriching personal and professional experiences that can not be included in the thesis, but are fundamental pieces of it, indeed.

First of all, I would like to thank my thesis supervisor, Prof. André Schiper, for all he did so that this thesis was a reality, for all the effort he put into every page of every paper we have co-authored, for all the things I have learned from him, for all the support he gave me during my worst moments.

I would also like to express my gratitude to the members of the Crystall project: Prof. Uwe Nestmann, Dr. Paweł Wojciechowski, Olivier Rütti, and Rachele Fuzzati; for those long but fruitful discussions where I learned how typical problems in distributed systems can be seen from a "programming languages" perspective. I am particularly grateful to Daniel Bünzli, who volunteered to review Chapter 5, and without whom that chapter would certainly not exist.

I am also very grateful to the members of the jury, Prof. Dejan Kostić, Prof. Ricardo Jiménez-Peris, and Prof. Richard Schlichting, as well as the president of the jury, Prof. Martin Odersky. They did an excellent job at reviewing the thesis and provided me with many interesting ideas and remarks. My thanks also go to Luis Rodrigues for the help provided with Appia.

I had the opportunity to share wonderful moments with LSR members and former members: Prof. Xavier Défago, Prof. Fernando Pedone, Dr. Péter Urbán, Dr. Matthias Wiesmann, Dr. Stefan Pleisch, Dr. Arnas Kupšys, David Cavin, Yoav Sasson, Richard Ekwall, Fatemeh Borran, and Dr. Martin Hutle, just to mention some of them. We were a real team at work, when doing teaching- and research-related tasks; and a real friend gang when skiing, watching films, holding barbecues at the shore of the lake or drinking a beer at Satellite. My gratitude also goes to France Faille, our secretary, for being such a kind person and for all her logistic support: if you are new to EPFL and do not know how to do an administrative task, don't panic, just ask France, she'll know what to do. I will always miss those moments around 10:40 am every day, when the whole Lab sat around that table and chatted together.

My gratitude also goes to Annick Panchaud, my fiancee, for helping me with the French version of the abstract, and for her love and understanding, especially at the moments when the writing of the thesis got stuck, which happened to be

the saddest moments in my life. Also to my mother and my sister, for their love and support, and for having them on the phone almost every evening for the last six years; and to the rest of my family, as well as Elvira and Angelita, for their affection, and for helping my mother out at those moments where *I* should have been there to help her out. I would also like to thank Annick's family, for their sympathy and hospitality.

During my Ph.D., I could also enjoy the company of good friends who deserve mention. Whether in Spain: César Moriano, Vicente Hernández, Enrique Ruiz, Ricardo Galdón, Jose Expósito, Honorato de Andrés, Laura Ferrando, Joe & Rose Cooper, etc.; or in Switzerland: David Portabella, Marcos Pérez, Dr. Guillermo Barrenetxea, Dr. Anil Alexander, Josep Garriga, Núria Sánchez, Dr. Raquel Urtasun, Rodrigo García, etc.; they have provided me with the freshness of thinking I needed to confront any difficulties in everyday work.

And last but, definitely, not least, I would like to thank my father. Simply for being a perfect father, from the very moment I was born till April 29^{th} 2006, much better father than I will ever dream of being with my children, yet I will do my best. *Papá*, everything I achieved in this life I owe it to you. And I do know I will be able to thank you for it, in person, at the very end...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Context and Motivation

Dependability and Fault Tolerance. A disappointing reality in Computer Science is the fact that processes often crash. Computer systems deployed nowadays tend to be more and more complex, usually comprising a number of collaborating subsystems that operate at different physical locations. As complexity in those systems grows, it becomes more difficult to ensure their correct operation. At the same time, our society tends to rely more and more on those computer systems, utilizing them even for critical tasks, whose malfunction may have a high cost in terms of money or human lives. Hence the need for these systems to be *dependable*.

Traditionally, research on dependability has been performed using different methods that can be classified as *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting* [Lap92]. Fault prevention strives to avoid the occurrence of faults in the system even before it is deployed. Fault tolerance tries to carry on with normal system operation despite the presence of partial malfunctions. The goal of fault removal is to reduce the amount of faults existing in the system. Fault forecasting are methods for detecting the presence of faults before they cause any harm to the system. The focus of this thesis is on fault tolerance.

Replication and Group Communication. Among the existing techniques to achieve fault-tolerance, namely *checkpointing*, *transactions* and *replication*, the latter is the only one that does not need to roll back the execution when a failure occurs. Replication consists in deploying identical instances of a subsystem in such a way that the state of different replicas is kept updated during system operation. Thus, when a replica fails, the surviving ones can take over so that the external user can not tell whether there has been a failure or not.

Over the last decades, various replication techniques have been proposed in the literature, which can be classified in two main categories: *hardware-based* and *software based* replication. Hardware-based replication systems put the emphasis on carefully designed platforms with replicated hardware [Bar81, BGH87] as well

as operating systems customized to operate in the so-called lock-step mode: the operating systems at all replicas are exactly at the same point in their execution. The drawback of this approach, apart from its high price, is that usually failures do not occur independently: faults caused by programming bugs will occur at the same time at all replicas if they are at the same state (e.g., Heisenbugs). Software-based replication does not require special hardware or operating systems in lock-step mode. It operates at a higher level: replicas are normal OS processes located at different machines and they fail independently: 99.3% of software bugs can be masked by software-based replication [Gra86]. There are two well-established techniques to implement software-based replication: *active replication* and *passive replication*. *Group communication* is a middleware layer, placed below the replication technique and above the network subsystem, that provides the properties needed by software replication techniques. Among these properties, the most important (and the most difficult to achieve) are agreement properties [GP96]. At the core of agreement problems is the consensus problem, which turned out to be unsolvable deterministically in asynchronous systems if at least one process can crash [FLP85]. So, with this impossibility result, a completely asynchronous system model is not very useful in the context of group communication. Chandra and Toueg [CT96] have proposed to augment the asynchronous system with the concept of failure detectors, allowing consensus to be solved deterministically. They also provide an algorithm solving consensus and prove that their algorithm uses the weakest failure detector that can be used to solve consensus [CHT96]. The good feature of algorithms based on failure detectors is that only liveness properties are compromised if the failure detector does not behave as expected, but not safety properties (the ones that keep the algorithm correct).

Group Communication Toolkits. For many years, group communication prototypes were monolithic implementations of a set of inter-related group communication protocols [CZ85, BJ87, KT91, BvR94, DM96, MFSW95, Mal96, BDGB95, EMS95, JKN96, MMSA+96]. They were monolithic because, even if the the toolkit was implemented using standard modular features of modern programming languages (libraries, packages, etc.), the protocols implemented (i.e., the core of the system) were intermingled in such a way that the protocol interfaces were sometimes not clear, and there were many hidden dependencies between protocols.

Monolithic toolkits had a number of drawbacks with respect to the modular toolkits that followed them. One of these drawbacks was maintenance: any change to a protocol implied certainly some changes in other parts of the system, due to the high degree of dependencies among protocols. Another important drawback was the great difficulty to reuse protocol implementations in another prototype. It was difficult because the protocol code was not encapsulated in one single module; besides, its interface was not standard or explicit. Monolithic prototypes did not use a framework to integrate the different protocols, but rather standard facilities pro-

vided by programming languages: function calls and callbacks, global variables, etc.

In the 90s, a new generation of group communication toolkits appeared: they were layered [vRBC+93, vRBG+96, VRBM96, Hay98, Ban02]. Layering was the first step in achieving modularity in protocol design. A layered protocol toolkit consists of a set of off-the-shelf protocols structured as layers where each layer only communicates with its upper and lower neighbor layer. This approach was inspired by the ISO/OSI architectural model for computer networks [ISO96]. As layering is a form of modularity, these systems benefit from advantages that modularity entails:

- *Configurability*. As interactions and dependencies between layers are explicit, the set of properties offered by the stack can be easily customized by changing the set of layers in a configuration.
- *Adaptability*. In order to adapt to changing operation conditions, or changing needs of the application, we can include new layers (that implement new properties), or replace some layer by another that operates better in the new conditions.
- *Efficiency*. If the application does not need some properties anymore, the layers providing them can be removed in order to avoid the unnecessary overhead.
- *Reusability*. Layers developed for some toolkit can be reused in other toolkits developed in the future.
- *Ease of debugging and maintenance*. Thanks to the full separation of layers, each one can be debugged and maintained independently. Moreover, maintenance is simplified since interactions between layers are explicit.

While layering represents an advantage with respect to monolithic systems, arranging all protocols in a strict stack is sometimes too rigid an approach. As we go up in the stack, the properties provided by protocols are more and more complex, and these protocols may need to interact with other protocols that are not necessarily neighbors (an example can be found in [PMR01]). Therefore, the trend has been to allow more and more flexibility in the way protocols are composed, thereby allowing several layers to coexist at the same level [Pin01], or allowing protocols to directly interact with any other protocol in a non-hierarchical fashion [HS98, WHS01, MPS93].

Protocol Composition Frameworks. A protocol composition framework is the infrastructure that allows a programmer to build a complex protocol (or service) out of a set of off-the-shelf building blocks. For the external user, the composition is perceived as a whole (large) middleware providing the expected (complex) service. In the same way as the goal of a middleware is to make the development

of distributed applications easier, the goal of a protocol composition framework is to make the development of a complex middleware easier. When monolithic group communication toolkits evolved to modular designs, the need for protocol composition frameworks became evident.

Many protocol composition frameworks in the 90s were ad-hoc: when a modular protocol suite was developed, this also included defining the rules of interaction between protocols. In other words, the same team that developed a new protocol suite also created a new protocol composition framework with the sole purpose of composing protocols of the new protocol suite. In some cases, they were so tightly coupled that it was difficult to determine the border between framework and protocol suite [Hay98, Ban02]. The clear distinction between a protocol framework and a protocol suite is crucial if we are to avoid the waste of effort that implies coming up with a new framework design every time we need to build a protocol suite.

Most relevant protocol composition frameworks in use are not ad-hoc, but general-purpose [MPR01, HS00, WHS01, MDB01, BGT+01, WRS04]. Hence, they are not biased to a specific set of protocols. They are general-purpose: their design and architecture does not make assumptions on the kind of protocols that are to be composed. They only set up the rules for protocol composition and interaction: who can interact with whom, how the interactions take place, what information is exchanged, whether interactions are synchronous or asynchronous, how concurrency is handled, etc. Moreover, because these frameworks are general-purpose, virtually any set of distributed algorithms with a modular design can use them.

Motivation. There is a broad diversity in the protocol composition frameworks in use nowadays. Sometimes, the design and behavior change radically from one framework to the other. Even terminology-wise, although there are some terms (e.g. *event*) that are widely accepted, the mechanism implied by a term can vary widely from one framework to the other. As a corollary, no framework is commonly accepted as the best one. Any attempt to defend a framework as being the best finds immediately strong arguments pointing out its weaknesses compared to some other framework. There is somehow a tendency to trust blindly one's favorite framework without wondering whether it is the best suited for the particular protocols under development.

This lack of agreement about which protocol framework should be used greatly reduces an important advantage of modularity: *reusability*. One can rarely reuse the protocol code developed for a framework F in another framework F' (an exception is Fortika, see Chapter 9). Due to the diversity of frameworks, protocol reusability is usually confined to the set of protocols developed by the same team. The ultimate consequence is having to develop from scratch protocols at all levels (even basic ones like "reliable point-to-point message transmission") every time a new group communication protocol suite is implemented.

On the other hand, group communication papers usually present and explain

new algorithms and prove them, but they seldom give a detailed description of the way the presented algorithm interacts with the application or with the lower-lever algorithms. It often remains a mystery how to execute lower-level primitives like *send* or *receive* found in the code: should the same thread execute the protocol code *and* the code of primitives? Or should it rather be a rendezvous *à la* Ada? No direct answer is usually found in papers on group communication.

In essence, this thesis demonstrates how modular group communication protocols can benefit from protocol composition frameworks, as well as how frameworks can be improved by looking at the protocol modules that use them. Therefore, the basic claim is that research in modular group communication and protocol composition frameworks must be addressed together.

1.2 Overview of Contributions

As we have seen, group communication toolkits used nowadays are modular. Thus, a good symbiosis between the protocol composition framework and the group communication toolkit is a crucial issue. In this thesis, we have done our research following these two complementary paths: protocol composition frameworks and group communication protocols. We believe this is the best way to proceed given that the strong need for modularity of current group communication protocols makes it difficult to ignore the underlying mechanisms to compose protocols.

We next give a bird's-eye view of the main contributions of this thesis.

1.2.1 Protocol Composition Frameworks

Perspectives for Framework Description and Comparison. There are different ways to look at protocol composition frameworks, as well as different features that can be present or missing. We propose four basic perspectives to describe and compare frameworks that constitute the basis of our methodology for research on frameworks. These perspectives are: composition model (how the components are arranged), interaction model (how the components interact), concurrency model (how the concurrency is managed within the framework), and interaction with the environment (how the application or the network interacts with the framework and vice-versa).

Comparing Appia and Cactus. When confronted to the dilemma of choosing the protocol composition framework that is best suited for our needs, usually there is no best choice. We demonstrate this by comparing Appia and Cactus, two protocol composition frameworks with a very different design. Overall, we can not tell which of these two frameworks is best. However, a thorough comparison using the four perspectives mentioned above shows that Cactus's interaction model is better while Appia's concurrency model is better.

Analyzing and Improving the Concurrency of Frameworks. Concurrency in protocol composition frameworks is an important issue; in particular, the question of managing concurrent threads that execute protocol code so that they do not run into race conditions or deadlocks. We survey the concurrency model of eight protocol composition frameworks and propose new features to improve the way concurrency is controlled.

The Header-Driven Interaction Model. In almost all protocol frameworks, the building blocks use *events* (see Sect. 2.1) as the basic mechanism to interact with other building blocks. However, it appears to us that events are not as good as one may expect. We describe the drawbacks of events and propose an alternative interaction scheme that uses message headers instead of events: the header-driven model. We show how the new interaction scheme overcomes the drawbacks of well-known event-based schemes.

1.2.2 Modular Group Communication

A New Architecture for Modular Group Communication. The architectures of group communication toolkits that appeared during the years are quite different among them. Nevertheless, they all share some common features. We discuss such features and the typical problems they entail. Then, we present a novel modular architecture for building a group communication toolkit. The new architecture is less complex, more powerful, and more responsive to failures.

Atomic Broadcast in the Crash-Recovery Model. *Crash-recovery* is a model where crashed processes can be restarted and resume operation at some point before the crash. To do so, crash-recovery protocols log process states to disk periodically. We argue that recent specifications of atomic broadcast (an important group communication primitive) that can be found in the literature are not satisfactory. We propose a novel specification that clarifies the properties that an application can assume from a crash-recovery atomic broadcast protocol. We propose two implementations of atomic broadcast and compare their performance with other well-known protocols.

1.2.3 Fortika

The *Fortika* Group Communication Toolkit. Fortika is the main prototype of the thesis. It is a group communication toolkit, written in Java, that can be composed using third-party frameworks like Cactus or Appia. Fortika contains a proof-of-concept implementation of the architectures, models and algorithms proposed in the chapters that follow: Fortika is the point of convergence of the contributions of the thesis.

Assessing the Crash-Failure Assumption using Fault Injection. The software-based fault injection technique consists in sporadically flipping bits in a message or an address in memory. These techniques intend to model spurious data corruption in real systems. We applied fault-injection techniques to our Fortika prototype to assess its tolerance to this sort of faults. The results were valuable to improve some aspects of Fortika's design.

1.3 Structure of the Thesis

The thesis is divided into three main parts. Part I is devoted to protocol composition frameworks. The results of this part are orthogonal to modular group communication, which is the subject of Part II. Part III acts as the convergence point of the two previous parts and is devoted to the Fortika toolkit. Finally, Chapter 11 summarizes the results of this works and suggests future research directions.

Part I

Advances in Protocol Composition Frameworks

Chapter 2

Protocol Composition Frameworks

Part I of this thesis is devoted to our research on protocol composition frameworks. In principle, the research we conducted on protocol composition frameworks is mainly in the context of group communication protocols, which is the research subject of Part II. Nevertheless, most contributions to protocol frameworks contained in this part are also valid in the more general context of distributed applications.

This is an introductory chapter to protocol composition frameworks: we present the basic terminology for protocol composition frameworks, as well as a concise description of the most relevant protocol composition frameworks out there. In short, this chapter contains the basic information that is necessary to fully understand the remaining chapters of Part I. Indeed, there are frequent references from those chapters to the present one.

2.1 Terminology

The lack of standard terminology to name the concepts of protocol composition frameworks motivates this section. Here, we unify the terminology of all protocol composition frameworks, with respect to the most important concepts. This is necessary in order not to wear out the reader with many terms that, while meaning the same, are different across frameworks. Moreover, sometimes the same term means different things depending on the framework considered. In the rest of this thesis, we will use the terminology defined here unless we are describing details of a particular framework, or a particular term or concept has no equivalent in any other framework. In any case, the text will be unambiguous regarding the definition used.

2.1.1 Composition and Protocol Modules

The code that implements protocols in a framework is organized into units called *protocol modules*. Protocol modules are software components, i.e., it is clear what services they offer to other protocol modules, and they usually use a few standard communication mechanisms (defined by the framework). The basic promise that protocol composition frameworks (and general component frameworks) make is that people other than the programmers of protocol modules are able to plug them together, without having to know or change the code inside the modules. We call such people *protocol composers* (or simply composers). The set of all protocol modules and their interconnections is called a *composition*. Every process in the system contains a composition. A composition can be either (1) a *stack*, if there can only be one protocol module (called *layer*) at a given level, or (2) a *graph* if protocol modules are arranged in a more flexible manner. *Composition time* is the moment when protocol composers statically put protocol modules together to form a composition. This is done before the composition is compiled and executed, therefore composition time occurs earlier than compile and run time. Protocol modules contain their *protocol state*. Certain frameworks allow a locally *shared state* among several protocol modules. Some allow even free access to the state of other protocol modules. This constitutes indeed a means of communication between protocol modules.

2.1.2 Asynchronous Communication

The main form of communication between protocol modules is asynchronous (even for local communication). The data transferred during an instance of communication is called an *event*. Initiating the communication is called *triggering* an event; the initiator module is called the *triggering protocol module*. Triggering an event results in *handling* the event: some piece of code, called *event handler* (or simply *handler*), is executed; the *handler* is part of the *handling protocol module*.

The fact that communication is asynchronous means that (1) handling the event may take place long after triggering is finished, (2) triggering is non-blocking, and (3) triggering never returns data from the handler executed (such as a return value or an exception that the triggering protocol module is expected to handle); but, of course, the handler module might trigger another event back to the initiator if such communication is necessary.

In our terminology, an event only exists for the duration of the communication. If executing an event handler results in communication with another protocol module, we consider the associated event a new event that is different from the original one. Of course, the new event might carry the same information as the original event. Some frameworks recycle events for more than one interaction, but this does not bring any conceptual difference. Sometimes, events are classified into *event types*. All events of a given type have the same signature, and convey the same kind of information. Event types are used for deciding which handler(s)

will be executed when an event of a given type is triggered.

Let us put asynchronous communication into context. Asynchronous communication between protocol modules is widespread because it maps to the message passing model of communication over networks. It often allows for a greater decoupling of protocol modules than synchronous communication, just like message passing allows for greater decoupling than remote procedure calls. The focus on asynchronous communication is an important feature that distinguishes protocol composition frameworks from general component frameworks.

2.1.3 Synchronous Communication

Some protocol composition frameworks offer synchronous communication. An instance of communication is called a *call*. A synchronous call is blocking and returns a value (or exception) that the calling module is supposed to handle. Sometimes these synchronous entry points are not implemented in protocol modules, but in the framework itself. In this case they are called *framework libraries*.

Most programming languages offer synchronous communication as their basic communication mechanism. Hence synchronous communication is easy to use for programmers.

2.1.4 Communication over the Network

Protocols and distributed applications have code in distinct address spaces. We call these address spaces *processes*. The communication mechanisms we have presented above are used within a process, and do not work between processes: a networking library must be used. Communication between processes can also appear as special protocol modules that do communication over the network.

Frameworks usually define a special data structure holding the information to be transmitted over the network. These structures are called *messages*. Messages have an optimized interface that allows to push and pop data units to and from the message. These data units pushed/popped by the framework are called *headers*, whereas the data pushed/popped by the application is called *payload*. Most frameworks push data in last-in-first-out fashion, others allow more flexible patterns. Because they are supposed to be transmitted on the wire, messages can also offer marshaling facilities.

The link between messages and events is that the latter convey the former from one protocol module to another in the same process.

2.1.5 Symmetric Compositions

Most real-life implementations of distributed algorithms have a curious property: the set of protocol module instances, as well as the way they are composed are exactly the same at all processes. This is called *symmetric compositions*. In this context, *peer protocol modules* are those protocol modules, one per process, that

are at the same place in the composition. Usually, peer protocol modules talk among themselves to provide the expected service.

A particular recurrent interaction pattern in symmetric compositions is called *peer interactions*: a protocol module communicates with its remote peers using the service offered by a lower-level protocol module. Peer interactions have always been the core idea behind protocol stacks (a universally known example is the ISO/OSI architecture [ISO96]). The purpose is to build protocols as incremental layers of abstraction: a protocol module, together with all its peer modules, implements a distributed service that is in turn used by the upper neighbor module (and its peers) to provide a higher-level service, and so forth. Figure 2.1 depicts an example of a peer interaction: the left part represents the conceptual information flow from A to its peer protocol A'. However, A needs that the message reaches its peer with certain guarantees, which the network subsystem is not able to provide in this example. Instead, A uses a lower-lever protocol module B (and B's peer, B') whose mission is to provide the guarantees A needs for its messages. Hence, the actual path that the message follows is the one depicted in the right part of Fig. 2.1.

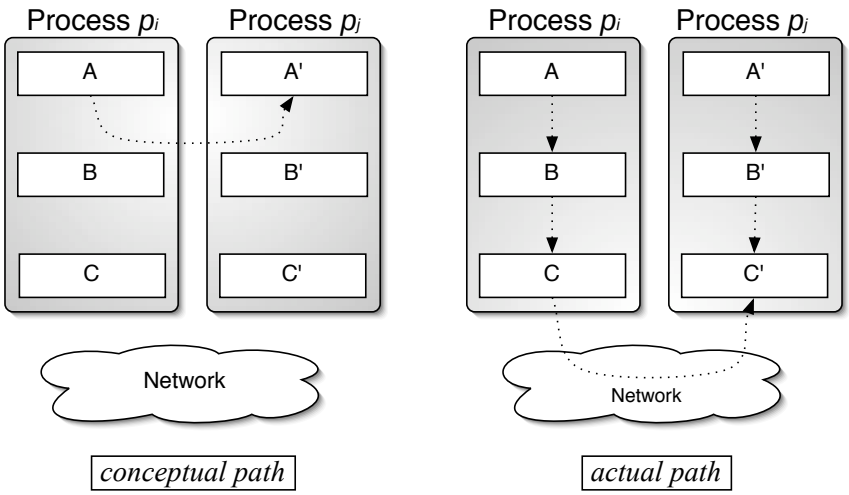


Figure 2.1: Example of peer interaction from A to its peer A'

2.2 Perspectives for Framework Description and Comparison

Describing and comparing protocol composition frameworks can be done from a number of viewpoints. An exhaustive analysis of such frameworks should cover the most relevant viewpoints. Here, we define four perspectives that allow us to describe the most important features of a framework. The four perspectives, together with performance, can also be used for framework comparison.

Composition Model. Specifies how protocol modules are arranged when they are composed. It can be hierarchical (protocol modules form stacks) or cooperative (no hierarchy). More complex models (e.g., a hybrid approach) are also possible. Some frameworks allow protocol *grouping*, where coarse-grain protocol modules can be made out of finer-grain ones.

Interaction Model. Defines the way protocol modules can interact and exchange information. It can be event-driven or it can additionally allow data sharing. If it is event-driven, a set of event types must be defined.

Concurrency Model. Describes whether and how concurrency is allowed in the framework. If some concurrency is allowed, the model should also specify how to synchronize concurrent threads. An interesting property that the concurrency model may provide is the so-called *FIFO event order*: events are handled in the same order they are triggered (i.e., events do not overtake each other).

Interface with the Environment. Describes how the composition interacts with the outside world, i.e., application, network and system resources (such as timers, etc.).

These four perspectives constitute the backbone of the remaining chapters of Part I. In the present chapter, we use these perspectives to describe most relevant protocol composition frameworks. Chapter 3 uses all four perspectives as a means to compare two different frameworks. Chapter 4 focuses the comparison on the concurrency models. Finally Chapter 5 focuses on the composition and interaction models.

2.3 Relevant Protocol Composition Frameworks

There is a big diversity of protocol composition frameworks nowadays. This section is an attempt to report on those frameworks that have achieved a certain popularity. We give a rather detailed description of the most representative ones, and a brief overview of the rest. We make an effort to present every framework using its own terminology and, at the same time, mapping such terms to our unifying terminology presented in Section 2.1.

2.3.1 Appia

In [MPR01], its designers define Appia as *a protocol kernel that supports applications requiring multiple coordinated channels and offers facilities for the application to express inter-channel constraints*. Appia is fully written in Java.

Composition Model. In Appia, a protocol module is defined as a pair *layer-session*. The layer defines three sets of events, namely, events *accepted* by the protocol module, events *provided*, and events *required* for proper operation. The session contains a private state and the protocol code. The latter is structured as a set of event handlers.

At composition time, a *Quality of Service* (QoS) is defined as a sequence of layer instances (see dashed squares in Fig. 2.2), and the framework carries out a correction check: a QoS is rejected if some layer instance declares an event as required, but no layer declares it as provided. Once a QoS has passed the correction check, it is used for session instantiation: for each layer instance in the QoS, the corresponding session is instantiated. This yields a sequence of session instances called *channel*. Finally, an Appia stack contains one or several of these channels. Two different channels may share a session at some level (see for instance sessions p and s in Fig. 2.2).

Interaction Model. The interaction model among protocol sessions is event-driven. It is important to point out that Appia events are designed for recycling events from session to session. Events are triggered by instantiating the event Java class representing the appropriate event type, and providing three parameters: the channel, the *source* session (i.e., the session that is triggering the event), and a *direction* (either upwards on downwards). These three parameters define the route of the event (i.e., the sequence of sessions). If a protocol layer did not declare a given event class as "accepted", its companion session will not be put into the event's route. Thus, all events of that type will bypass this session. A session forwards an event to the next session in the event's route by calling the event's method *go()*. Events convey data inside. Data sharing among sessions is not possible.

The framework has adopted a so-called *open event model* [MPR01], which means that new event classes may be defined, in contrast to the *closed event model* of its precursor Ensemble [Hay98]. New event types are defined by inheritance: the root event is class *Event*, and any other event type inherits from it or from one of its descendants. The advantage of the open event model is that legacy protocol modules can deal with new event types, regarding them as an ancestor class (inheritance-controlled polymorphism).

Events are unable to get out of the channel they are in: when they end their channel route, they are destroyed. There is one exception: *SendableEvent* (and its descendants). Sendable events contain a *message*: at the bottom-most protocol module, the message is marshaled and transmitted to its destination; at the top-most protocol module, the extraction of the message is treated in an ad-hoc manner. Messages have a simple interface consisting of *pushing* and *popping* headers.

Concurrency Model. All events in all channels in the stack are processed by one single thread: the event scheduler thread. The main advantage of this single-threaded model is the absence of racing conditions inside sessions' code, i.e., pro-

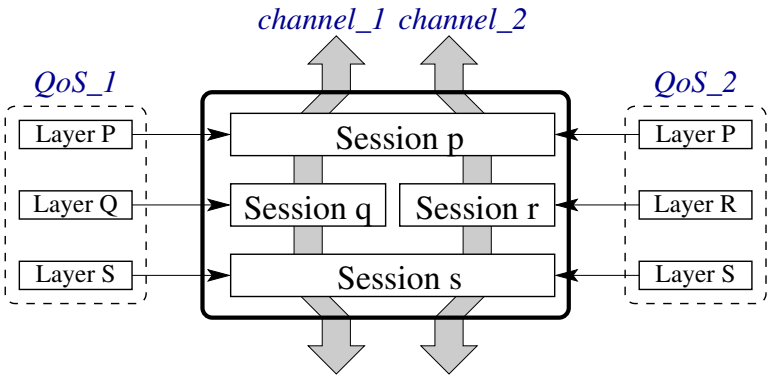


Figure 2.2: Composition example in Appia: Stack with two channels. Both channels share sessions *p* and *s*.

tocol developers never have to worry about thread synchronization. All events are put into the event scheduler's queue. In each step, the event scheduler pops the first event e from the event queue, looks up the next session that handles e, and executes the corresponding event handler. If the session creates a new event e' in the same direction as e, e' is inserted in the event queue immediately after e. Otherwise, e' is inserted at the end of the event queue. This way of queuing up events is further discussed in Chapter 4.

Interface with the Environment. The outside world can insert events into a channel to notify about external happenings (e.g., a packet has arrived from the network, the application has sent a message). For this purpose, Appia provides a special mechanism that external threads must use in order not to bring on race conditions. This mechanism is called *asynchronous events* in Appia[1] and is the only way for the application and the network to interact with an Appia stack.

As a minor interfacing detail, Appia defines a special event class called *TimerEvent* that sessions can use to set up timers.

2.3.2 Cactus and the x-kernel

In [HP91], the x-kernel was defined as a *kernel designed to facilitate the implementation of efficient communication protocols*. The x-kernel design had visionary ideas as, for instance, allowing graph-based rather than stack-based composition (which is the current trend). Ten years later, Cactus was in turn defined in [WHS01] as *a framework for constructing configurable protocols and services, where each service property or functional component is implemented as a separate module*.

[1] Unlike the rest of events (which can only be triggered when handling another event) they are called *asynchronous* because they can be triggered at any moment.

Due to the fact that Cactus is an extension of the x-kernel, it is not possible to describe Cactus without describing the x-kernel on the way. Thus, we describe the two systems together. The x-kernel was initially written in C, but other versions have appeared later. Cactus has versions in Java, C and C++.

Composition Model. Cactus defines a *two-level* composition model: coarse grain and fine grain. The coarse grain level is inherited from its ancestor, the x-kernel. The coarse grain protocols, called *composite protocols*, are composed by defining a hierarchical graph. In this graph, several composite protocols may be placed at the same level.

This is the only way to compose protocols in the x-kernel: the coarse-grain protocols are themselves monolithic entities. In Cactus, the composite protocols enclose a set of finer grain *micro-protocols* arranged in a cooperative way (no hierarchy) and interacting via event triggering and data sharing. Micro-protocols cannot exist on their own in the hierarchy graph: they need to be within a composite protocol.

Interaction Model: Composite Protocols. Composite protocols are linked by edges that define an directed acyclic graph. Only composite protocols that are linked by an edge in this graph can directly interact. Composite protocols are instantiated at run-time yielding *sessions*. Each composite protocol session maintains its own state and its own micro-protocols. A session can *open* a new session of a composite protocol to which it is linked. The communication between two sessions is by *message* passing. In the x-kernel, messages are structured as a stack of headers, with the usual *push* and *pop* operations. Cactus extends the mechanisms for dealing with messages. In Cactus, a message is a data structure consisting of a set of *attributes*. Attributes are tuples of the form *(tag, scope, value)*. *Tags* are used to retrieve the attributes' *values*, which are the actual data. The *scope* restricts the attribute's visibility. A session may send a message up or down to the next session in the graph. When sending a message up, if there are several sessions that may receive the message, a *demux* function must be provided in order to decide which is the receiving session (this decision is usually based on some attribute conveyed by the message).

Interaction Model: Micro-Protocols. As previously said, micro-protocols only exist in Cactus. In the x-kernel, the coarse-grain protocols are themselves monolithic. Micro-protocol execution is event-driven: micro-protocols are structured as a set of event handlers and contain private state. The protocol code is contained in the event handlers, which can modify this state and trigger other events. Micro-protocols can also share state (see Figure 2.3).

At start-up time, a micro-protocol *binds* its handlers to event types. Every time an event is triggered, a structure called *event occurrence* is created to carry the

data associated with that event. Upon triggering an event e in some composite protocol C, the framework will execute all handlers in all micro-protocols of C that are bound to e's event type, giving each of them the event occurrence created. A priority is chosen at binding time to set the execution order of all handlers bound to e's event type within a composite protocol. Events do not cross session boundaries: they are only seen by micro-protocols within the session in which the event was triggered. An important feature of Cactus is the ability to bind/unbind event handlers to events on-the-fly, which can change the behavior of a micro-protocol at run-time. Moreover, a micro-protocol may be activated or disabled on-the-fly (by binding/unbinding all its handlers).

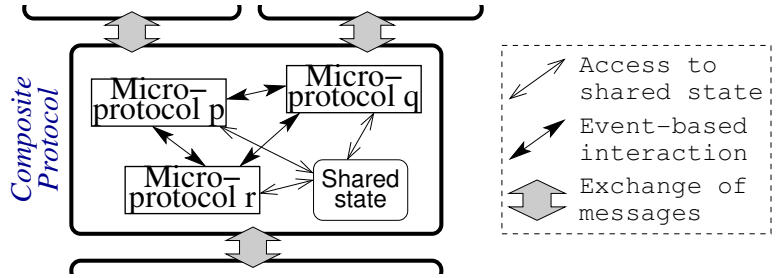


Figure 2.3: Example of Cactus composite protocol.

Concurrency Model. In the x-kernel, the approach to concurrency is the so-called *thread per message*. In such an approach, the same thread will shepherd the message's way through all protocols that handle it, i.e., the different messages that reach the composition are handled by different threads, which permits that all of them are processed in parallel. There is no built-in mechanism provided by the framework to limit this concurrency in case of racing conditions. Therefore, protocol programmers have to synchronize threads accessing critical resources on their own.

In Cactus, a composite protocol, i.e., all the micro-protocols it contains, is idle as long as no message arrives from above/below. Message arrivals start out the activity in a composite protocol. At that point, the framework triggers an event of the predefined type *"message has arrived"*: all handlers of micro-protocols bound to this event will be executed one after the other.

The handler code can modify the private state in the micro-protocol as well as the shared state in the composite protocol, and can trigger other events. There are two ways to trigger an event: *raising* and *invoking*. *Raising* is asynchronous (a thread is spawn or reused from a thread pool), while *invoking* is synchronous (what we know as *calls* in our terminology). In the *raising* scheme, the private state of a micro-protocol may be exposed to racing conditions. So, it is necessary to synchronize the access to this private state (as well as to the shared state in

the composite protocol). Such a synchronization is not enforced by Cactus (like in the x-kernel); it is the responsibility of the protocol programmer to enforce synchronization using standard mechanisms. Yet, some versions of Cactus (e.g., C versions) enforce, by default, some basic synchronization properties such as atomic handler execution.

Cactus defines a special way to trigger events: *coordinated raising*. This is done using the *register* and *signal* methods. When a set S of micro-protocols mp_i execute *register(*mp_i, e*)*, where e is an event type, then the event will only be triggered once *all* micro-protocols mp_i in S have executed *signal(*mp_i, e*)*.

Interface with the Environment. There are no restrictions for Cactus micro-protocols or x-kernel protocols to use OS resources as timers, sockets, etc. For timers, Cactus offers the possibility of raising an event after some delay. As for the network, the micro-protocols at the lowest level may open sockets themselves or can be placed on top of an x-kernel transport stack. The application, which is usually placed on top of the hierarchy graph, has no special interface: it has to use an interface as though it were just another composite protocol.

2.3.3 Samoa

In [WRS04], Samoa is defined as a *protocol framework that ensures the isolation property*, as defined in the context of databases. Samoa *has been designed to allow concurrent protocols to be expressed without explicit low-level synchronization, thus making programming easier and less error-prone.* Samoa is an alive, evolving project. Here, we describe the features of Samoa that were already present in [WRS04]. The current prototype is written in Java.

Composition Model. Protocol modules are composed in Samoa in a cooperative way, i.e., non hierarchical. This compositional scheme is quite similar to Cactus at its fine-grain level.

Interaction Model. In Samoa, protocol modules are composed of a set of event handlers and a local state. Unlike Cactus, this local state is confined to its protocol module, and not visible from other modules. This avoids hidden dependencies that reduce the advantages of modularity. Event handlers are bound to event types and they are executed whenever an event of that type is triggered. Every event belongs to a predefined event type. The handler code typically modifies the local state and triggers new events, which will cause the execution of all handlers bound to their event types. Protocol execution is purely event-driven: the code in a protocol module is only executed in response to triggering an event (i.e., protocol modules do not contain threads).

There are two kinds of event types: internal and external. Internal events can only be triggered by handlers, whereas external events typically represent external

requests (message arrived from the network, message sent by the application, etc) or asynchronous requests (timeouts). The reason for this distinction will become clear when we explain the concurrency model.

Finally, the binding from handlers to events can be modified at runtime. This allows for instance a protocol module to be deactivated at a given point in the execution and a new protocol (or a new version of the same protocol) to take over from that time on. This is known as *protocol switching* and can be exploited when adapting to changing conditions (e.g., network congestion) is needed. When the replacing protocol is indeed a newer version of the protocol replaced we call it *dynamic protocol update*. Samoa has allowed research on *coordinated protocol update*: a protocol module is updated on the fly *at all processes*, and this is done without breaking down the protocol semantics (i.e., protocols are updated transparently to the application, which is never stopped).

Concurrency Model. The Samoa framework presents an advanced concurrency model that makes it unique. On the one hand, it gives the illusion to the protocol programmer that the framework provides no concurrency. This frees the protocol programmer from the burden of low-level synchronization and the risk of race conditions. On the other hand, the actual execution is indeed concurrent, but the framework scheduler makes sure that the state resulting from that concurrent execution is equivalent to some sequential execution. We now explain concisely how this is achieved.

We say that all events triggered by a handler executed in response to event e causally depend on e. This causal dependency relation is transitive and reflexive. When an external event is triggered, the handlers bound to its type are scheduled for execution. This marks the start of a new *computation*, which includes the initial external event and all internal events that causally depend on it. Informally, a computation can be regarded as a kind of transaction. The Samoa runtime system provides the *isolation* property to computations: when two or more computations are executed concurrently, the resulting state of all protocol modules must be equivalent to some sequential execution of those computations. This *isolation* concept is analogous to that of transactions in databases, but without atomicity, consistency, and durability properties, which are not needed in this context.

Summing up, Samoa is able to execute protocol code with a certain degree of concurrency, while giving the impression to the protocol programmer that no concurrency takes place. This makes protocol programming easier and less error-prone.

Interface with the Environment. The environment interacts with Samoa by means of external events. We have seen above that the special role of external events is important for managing concurrency. Timers are also represented as external events, with the advantage that protocol modules do not own internal threads to implement timeouts, which would break a purely event-driven model.

As for interactions from the composition towards the environment, there are no special provisions made by Samoa. Nevertheless, the mechanism chosen should not block the computation (e.g., in an I/O operation) for a long time.

2.3.4 Other Protocol Composition Frameworks

Apart from the protocol composition frameworks presented above, which are the most relevant for this thesis, there is a big diversity of other frameworks that should be concisely presented.

Neko. Neko [UDS02] is an environment to test prototypes of distributed algorithms and assess their performance. It is written in Java and its key feature is that the same implementation of an algorithm is used for simulation and for real testing (with no need to adapt the code or recompile it), which saves precious time in building and assessing prototypes.

Protocol modules are composed in Neko in stack fashion, following a typical hierarchical model. Thus, protocol modules are called layers. The way layers interact with their neighbor layers is by synchronously calling a special method that all protocol modules must implement (we can consider this method as the singleton handler). Events are not used, since the handling method is directly called from the initiating layer. It is possible to define *active layers*: layers that own threads. This allows concurrent execution, which is managed in an ad-hoc manner by the protocol code.

Neko comes along with an extensive set of distributed algorithms, as well as tools and scripts for performance evaluation. Chapter 4 of this thesis contributed to extend Neko's protocol composition capabilities.

Eva. The Eva *event-based framework for developing specialized communication protocols* [BGT+01] is written in Java and was used to build the Eden group communication toolkit.

Eva defines a hybrid composition model where a set of cooperative protocol modules can be grouped and seen as a single *component*. The lowest level components (i.e., those that are not composite) are called *entities*. The same entity can be put inside two different components. Eva is an event-driven framework, and uses the concept of *event channels*. Entities within a component subscribe to an event channel, which acts like a *bus* conveying events from the triggering protocols to the handling ones. A given event channel is only visible within the component where it has been defined. Therefore, the grouping feature in Eva is mainly used to route and filter events: an entity can *see* (and handle) an event e from outside only if its enclosing component also handles e.

Eva allows for multithreading, although nothing is provided to control the concurrent execution of the whole composition. So the programmer has to manage concurrent threads using standard Java synchronization.

Ensemble and JavaGroups. Ensemble [Hay98] and its Java re-implementation JavaGroups [Ban02] (also called JGroups recently) are not protocol composition frameworks, but group communication toolkits. However, as their protocols are modular and composable, they need to use an ad-hoc set of rules for protocol composition. Here, we only describe these rules, which have no special name since they are not presented as a protocol composition framework. This makes it sometimes hard to define the boundary between framework and protocols. The way protocols are composed in Ensemble is very similar to Appia. Indeed, Appia was born as a proof-of-concept implementation in order to fix certain limitations in Ensemble [PMR01].

Ensemble uses a fully hierarchical composition model, where compositions are a stack of layers. This stack is strict in the sense that only one protocol module can operate at each level of the stack. The interaction model is event-driven, where events flow up or down the stack looking for layers that handle them. The set of event types is fixed, and it is not possible to define new types or extend the existing ones. This can be explained by the fact that it is an ad-hoc framework for group communication, but it also hinders the development of new modules for group communication, since they have to get along with the already existing event types. Finally, Ensemble is single-threaded, containing one scheduler that treats one event at a time and shepherds that event all the way up (or down) the stack.

A distinctive feature of JavaGroups is its nice design for the application-stack interaction. It defines a set of off-the-shelf interaction patterns, called *building blocks*, which the composer can choose from. Consequently, the chosen building block is placed between the stack and the application. Thus, the application can decide the way it interacts with the stack. Examples of building blocks are "message queues" or "callbacks".

SDL. The Specification and Description Language (SDL) [EHS97] is not, in principle, intended as a runtime framework for protocol composition. It is rather a ITU-standardized language, widespread in the telecommunications industry. Its main use is modular specification of communication protocols and hardware components. Nevertheless, it is suited for any application based on finite state machines, and any composition of them. Recently, some platforms implement a runtime module able to execute SDL code, which has sparked some interest in the use of SDL as a framework for modular software development.

An SDL system is organized hierarchically in *blocks*, which can contain other blocks. The blocks that do not contain other blocks but only protocol code are called *processes*.[2] Blocks/processes are interconnected through *channels*. Each channel can convey any number of *signals*. Signals can be seen as events, whereas channels are a sort of event binding between two or more blocks. It thus follows an event-driven model.

[2]Note that this definition of *process* is quite different from ours.

Protocol programming is based on extended finite state machines communicating through signal exchange, without any shared memory. The protocol programmer can define local variables as well as set conditions on the signals that are to be received and on the values of their parameters. Every process (i.e., protocol module) is programmed using one of such finite state machines (i.e., a thread of execution). All instances inside a block run concurrently. Several processes can be active at a time, which also means that several finite state machines can run concurrently. SDL also allows the definition of timers. When a timer expires, a signal is sent to the initiator of the timer. This timer signal is processed as an ordinary signal.

In short, SDL implements a special multithreaded concurrency model called *thread-per-layer model*. Its concurrency is limited: every SDL process contains a finite state machine (i.e., a thread) whose execution is confined to this process. The only way to interact with other SDL processes is by means of signal exchange. As a result, even if the degree of concurrency can be high (e.g., a system with many processes), no action needs to be taken in order to prevent deadlocks or racing conditions, since a given piece of code will never be executed by more than one thread.

SDL was used as the supporting framework for a modular group communication protocol in [EMPS04a, EMPS04b].

2.4 Roadmap to the Remainder of Part I

So far, we have set up a unifying terminology for the elements and concepts that can be found in a protocol composition framework. We have also introduced our four perspectives for framework comparison as a common way to analyze and compare them.

Once we have set up this introductory background, we can proceed with our contributions. This is done in the remaining chapters of Part I. In Chapter 3, we compare two of the most relevant protocol composition frameworks: Appia and Cactus. We compare these frameworks from the perspective of *all* the models for framework comparison defined in Section 2.2. The aim of this comparison is to shed some light on the frameworks' features: which are useful, which are unnecessary, and which are missing. A performance comparison is also presented.

One of the conclusions of Chapter 3 is that the concurrency model of Appia and Cactus can be improved. This is the motivation for Chapter 4, which surveys the concurrency model of a number of protocol composition frameworks (including Appia and Cactus). This chapter proposes several features to achieve a simpler and more efficient way to manage concurrency in protocol composition frameworks.

Finally, Chapter 5 copes with the composition and interaction models. In particular, a new interaction scheme for protocol modules is presented, which does not use events, but messages headers.

Chapter 3

Comparison of Protocol Composition Frameworks

In this chapter, we compare Appia and Cactus, two frameworks for protocol composition. The comparison, based on the experience gained in implementing a fault-tolerant Atomic Broadcast composition, covers the four perspectives for framework comparison defined in Chapter 2. This chapter also provides performance results, and concludes with a discussion of the most interesting features of the two frameworks, and suggestions for an improved framework.

3.1 Introduction

As discussed is Chapter 1, a protocol composition framework is the infrastructure that allows a programmer to build a complex protocol out of a set of off-the-shelf building blocks: the composition. The composition can be arranged in a stack or in a graph, which is more flexible. For the external user, it is perceived as a whole (large) middleware providing the expected (complex) service. In the same way as the goal of a middleware is to make the development of distributed applications easier, the goal of a protocol composition framework is to make the development of a complex middleware easier. As distributed applications become more and more complex, they require stronger guarantees from the underlying middleware.

Group communication middlewares are not an exception. At the beginning, group communication middlewares like ISIS [BJ87, Bir93], Transis [DM96], Phoenix [Mal96, MFSW95], etc. were monolithic: their protocols were tightly coupled and had plenty of hidden dependencies, even though they were based on well defined building blocks. The fact that they were monolithic prevented these building blocks from being reused elsewhere. Later on, group communication middlewares started to be modular: Consul [MPS93], and Horus [vRBG+96, vRBC+93] pioneered modular group communication. Finally, state-of-the-art group communication toolkits like Ensemble [Hay98], JavaGroups [Ban02] (also called JGroups), Appia's group communication toolkit [Pin01], or the group membership composite

protocol described in [HS98] are all modular: they are built from smaller off-the-shelf building blocks called *protocol modules*. Protocol composition frameworks yield a number of advantages over monolithic approaches, namely configurability, reusability, extensibility and ease of maintenance (see Sect. 1.1). However, they may yield plenty of drawbacks if they are not properly used (e.g., bad protocol module design, hidden dependencies left over, etc.), and might render a complex middleware protocol even harder to maintain than in the case of monolithic approaches.

The protocol composition frameworks for the development of modular group communication middlewares that we are aware of can be classified into two families. The first family consists of the x-kernel [HP91], and its successors Coyote and Cactus [Bha96, BHSC98, HS98]. The second family, is composed of Horus and its successors Ensemble, Appia [MR99a, MR99b, MPR01] and JavaGroups. The most representative frameworks in each of these two families are Cactus and Appia. Chapter 2 provides a detailed description of these two frameworks.

The Appia project has started after Cactus, but is already mature enough to be tested and compared with Cactus. In this chapter we analyze and compare these two frameworks, in particular their Java versions.[1] Apart from previous research using Appia and Cactus [Men01, WMS02a, WMS02b], we base our comparison on the experience gained while implementing a prototype group communication middleware (see Sect. 3.2), written in Java, using both frameworks.

The outcome of our comparison is that each framework has some unique interesting features, but none of them is clearly better than the other.

The rest of the chapter is structured as follows. Section 3.2 describes the prototype group communication composition that has been implemented both in Appia and Cactus. In Section 3.3, we compare these two frameworks. Section 3.4 contains suggestions for better frameworks. Finally, Section 3.5 concludes the chapter.

3.2 Composition Implemented

3.2.1 Description

In order to compare Appia and Cactus, we have implemented the same composition in both frameworks. This composition is part of the Fortika prototype (see Chapter 9) and provides fault-tolerant atomic broadcast in a static process group (see Sect 6.3.5). There are two reasons why we chose atomic broadcast for this implementation: (1) it is a fairly complex protocol with a set of well-known building blocks, and (2) it is instrumental for group communication: the context of Part II. Atomic broadcast has been implemented by the composition of several protocol modules (see Fig. 3.1, where arrows correspond to event types). A full description of these protocol modules will be given in Chapter 6, here we give a short description of them:

[1] In the case of Appia, the Java version is the only one we are aware of.

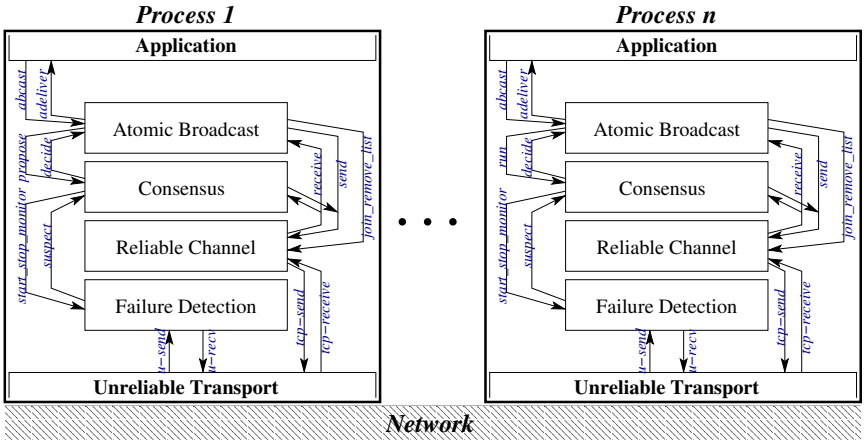


Figure 3.1: Protocol modules for Atomic Broadcast.

- *Reliable Channel*. This protocol module provides reliable message communication between two processes located on different machines. The interface consists of the event types *send* and *receive*, depicted in Figure 3.1. The protocol module also receives information about the group membership change (event type *join_remove_list* in Fig. 3.1). The current implementation is based on TCP.[2]

- *Failure Detector*. This protocol module implements a failure detector based on ping messages.

- *Consensus*. This protocol module implements an algorithm for solving consensus [CT96]. It interacts with the Failure Detector module by means of event types *start_stop_monitor* and *suspect* (see Fig. 3.1). It also interacts with the Reliable Channel module.

- *Atomic Broadcast*. This module implements Atomic Broadcast [CT96]. It interacts with the Consensus module (via the *run* and *decide* event types) and the Reliable Channel module.

3.2.2 Conforming to Fortika Conventions

We carried out this implementation in the context of Fortika (see Chapter 9), the reason being to implement the protocols for both frameworks as similarly as possible. Therefore, we have followed a set of conventions required by Fortika. We next give a brief description of such conventions and refer the reader to Sect. 9.2 for further details.

[2]We use Robust TCP connections [EUS02], which properly mask link failures.

3.2.2.1 Common Protocol Code for Both Frameworks

All the protocol modules of Figure 3.1 have an internal private state and consist of a set of event handlers that may read and/or modify the private state and trigger events. Each of these protocol modules has been implemented as a Java class in which the event handlers are public methods and the state is stored in private attributes. These *protocol classes* are instantiated in both composition frameworks without changing a single line of code (see circles in Fig. 3.2). As a result, the actual implementation of each protocol module is exactly the same in both frameworks.

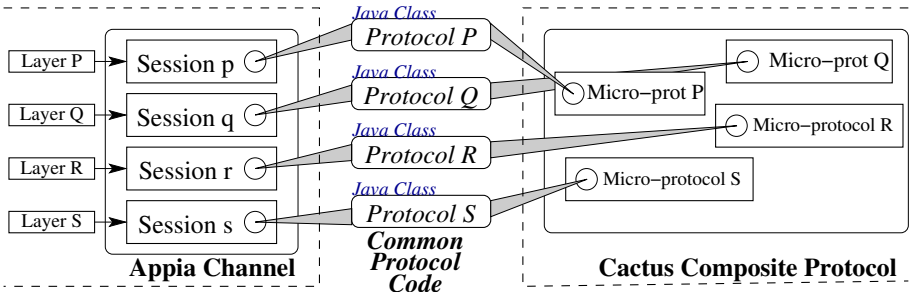


Figure 3.2: Protocol code shared by Cactus and Appia.

3.2.2.2 Event Routing Using Framework Facilities

We composed these protocol objects (i.e., these instances of protocol classes) using Cactus' and Appia's infrastructure:

- *Cactus:* As we can see in Figure 3.1, almost every protocol interacts with every other protocol. In Cactus events are unable to cross composite protocol boundaries. For this reason, we have used in Cactus one single composite protocol for the whole composition (see Fig. 3.2). Every protocol object has been wrapped into a micro-protocol, which binds handlers in order to capture events from the environment and routes them to the appropriate handler. The wrapping micro-protocol also intercepts events triggered within the protocol object, and translates them into a Cactus event invocation.

- *Appia:* In Appia, every protocol object has been wrapped within a protocol session. The companion layer declares all events that the protocol object provides, requires and accepts. The session code has the same role as micro-protocols in the Cactus version: event handling only consists of (1) calling the appropriate handler inside the protocol object, and (2) intercepting all events triggered inside the common protocol code and translating them into Appia events, which are routed along the channel by calling method *go()*.

3.2.2.3 Concurrency Model

We have implemented a similar concurrency model in both frameworks. In Appia there was only one choice, since the event scheduler is single-threaded. In Cactus, we have protected the composite protocol with a *mutex* to avoid concurrent execution. Moreover, we use (synchronous) event invocation to prevent spawning of new threads inside the composite protocol. This choice avoids to some extent the cost of context switches, and allows a fair performance comparison between Cactus and Appia.

3.3 Comparison

This section is devoted to a comparison between the two frameworks used to implement the composition described in Section 3.2. Even though the models adopted by Appia and Cactus are very different, they share some common features. These features are pointed out in Section 3.3.1. Then, we discuss all the issues that make these two frameworks so different (Sect. 3.3.2). Finally, a performance comparison concludes this section.

3.3.1 Similarities

3.3.1.1 Common Features

The Event-Driven Interaction Model. The internal structure of Cactus' micro-protocols and Appia's sessions is the same: a set of event handlers (with an internal state). These event handlers play three main roles: (1) managing messages, (2) modifying the internal state, and (3) triggering events that may cause handler execution in other protocol modules.

A protocol module does not have the initiative for execution. It is rather in an idle state, waiting for some happening to occur: an event. It then reacts to the event by executing one of its event handlers and triggering other events, which make surrounding protocol modules react to them

The fact that Appia and Cactus both implement the event-driven interaction model is the most important common feature for us, since without this resemblance it would have been impossible to share the protocol code in the implementation of our *Atomic Broadcast* composition.

Message Abstraction. Both frameworks have the concept of message. Messages are the only information able to reach the network. Although message structure differs, the concept of *headers* is present in both frameworks.[3] If a given protocol module pushes a header into a message, only its corresponding modules at other processes are able to access and strip off this header. As a message flows down to the network, protocol modules see upper modules' headers as application data.

[3] Cactus does not define headers, but attributes with *peer* scope behave exactly as headers.

Likewise, as a message flows upwards, each protocol module strips its header off the message. Our common protocol code benefits from this similarity between both frameworks, treating messages in an homogeneous way.

3.3.1.2 Common Lacks

Flow Control. Neither Appia nor Cactus have any form of flow control that would bound the workload (events in Appia, messages in Cactus) produced by the environment (network, application). In Appia, the absence of flow control may lead the event queue to overflow if flooded with incoming events (see Sect. 3.3.3). However, at the time we carried out this work, there was an ongoing effort to incorporate flow control to Appia, limiting the aggregate size (in bytes) of events in a channel.

In our Cactus implementation, the problem is prevented thanks to the *mutex* that protects execution in the composite protocol.

Efficient Message-Passing Network Interface. Another lack of both frameworks is the absence of a message-passing network interface built in the framework itself. There are some protocols included in both framework distributions that implement a message-passing interface for upper-level protocols (e.g., Appia has protocols like *UDPSimple*, *TCPSimple*, and Cactus has protocols like *UTP*), but when the user needs do not match what these protocols offer, she is forced to cope with sockets directly.

3.3.2 Differences

3.3.2.1 Composition Model

Appia's Channels *vs.* Cactus Cooperative Composition. Protocol composition is quite different in each of the frameworks. In Appia, the concept of channel yields a strictly hierarchical composition, even though channels are allowed to share sessions: events flow up and down following a dedicated channel. This restricts in Appia the possible ways in which protocol modules can interact. Consider for instance Figure 3.3(a), where event $e1$ goes from protocol module p to protocol module q, and then to r. Appia's channels are optimized for this case. Figure 3.3(b) shows a different itinerary for event e_2: first from p to r and then from r to q. This is much trickier to express in Appia, since this order clashes with the order defined by the channel.[4] Appia has a workaround consisting of using a special event called *EchoEvent*, which conveys an event and travels up to the top (or down to the bottom) of the channel. At the end of the channel the conveyed event is extracted and reinjected into the same channel, but in the opposite direction. In Figure 3.3(b), protocol p would put an event $e2$ inside an *EchoEvent*, and send the latter up.

[4] Using two channels does not solve the problem, since a given event instance is bound to only one channel.

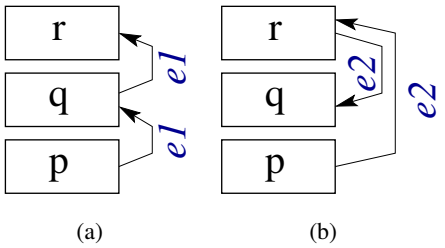


Figure 3.3: Two different patterns of event flow. Appia channels only support the event path depicted in (a). Figure (b) is only possible in Appia with the help of *EchoEvent*.

This problem does not exist in the non-hierarchical composition of Cactus, where any micro-protocol can send an event directly to any other micro-protocol. Handler priority in Cactus allows to express either the event flow in Figure 3.3(a) or in Figure 3.3(b).

Correctness Check. Cactus does not offer any correctness check. Appia does a composition correctness check at QoS creation time (see Sect. 2.3.1). This correctness check rejects some incorrect compositions, but does not aim at being complete: an incorrect composition may pass this check.

3.3.2.2 Interaction Model

Multiplexing Events. Some compositions have complex event handling constraints. Two different protocol modules may accept the same event type, but each event is for only one of the two modules. Consider for example Figure 3.1, where *Reliable Channel* triggers *receive* events. Every *receive* event is handled either by *Consensus* or by *Atomic Broadcast*, but never by both. The *receive* event must be handled by *Consensus* if the event is the result of a *send* event triggered by *Consensus* in the corresponding remote process; the *receive* event must be handled by *Atomic Broadcast* if the event is the result of a *send* event triggered by *Atomic Broadcast* in the corresponding remote process. In Appia, these two events travel through the same channel,[5] i.e., there is no way to reach *Atomic Broadcast* directly. In other words, all up-going *receive* events will first be handled by *Consensus*, even if they are not aimed at it.

In Cactus, micro-protocols can directly interact, but the problem is similar. Both *Atomic Broadcast* and *Consensus* bind a handler to event *receive*. Therefore, upon triggering of this event (by *Reliable Channel*) both micro-protocols will

[5] A two-channel solution may work, but with such a solution, Consensus and Atomic Broadcast *are* aware of each other, which is not desirable.

handle the event, which is incorrect. We describe a solution to this problem in Section 3.4.2.

As a conclusion, any interaction based on Appia's notion of channel can be implemented in Cactus in a straightforward way: the micro-protocol composition in Cactus is a generalization of Appia's channel-based composition.

Shared Data. In Appia, protocol interaction is only possible by means of events. Cactus offers data sharing among micro-protocols, in addition to event triggering. However, in order to reuse the same Java protocol classes in Appia and in Cactus, data sharing was not used in our Cactus implementation of the *Atomic Broadcast* composition. In this specific example, we do not think that data sharing would have been very useful.

In the general case, data sharing constitutes a dependency among protocol modules, whose access is usually hard to coordinate, especially if the protocol modules are designed by different people.

FIFO Guarantee. The *adeliver* events triggered by the *Atomic Broadcast* protocol module in Figure 3.1 have to be handled by the application (or by any other protocol module composed on top of *Atomic Broadcast*) in FIFO order, so as to preserve the total order provided by Atomic Broadcast. This is ensured by Appia, which enforces FIFO order among events (see Sect. 2.3.1). Cactus does not provide such a property. Nevertheless, in our implementation, the absence of concurrency within the composite protocol indirectly ensures the FIFO guarantee.

3.3.2.3 Concurrency Model

As already said in Section 2.3.1, Appia follows a single-threaded model, which prevents race conditions. In Cactus, the programmer needs to properly synchronize concurrent access to shared data within a micro-protocol, and within a composite protocol (if data is shared). A simpler solution is to consider a setting in which only one thread is allowed to execute in a composite protocol at a time (see Sect. 3.2.2.3). Such a design yields a concurrency model much closer to Appia in the sense that there is no need to include thread synchronization in the micro-protocol code. We will come back to this issue in Chapter 4.

3.3.2.4 Interface with the Environment

In Cactus, both the application and the network have to adopt the interface of a composite protocol, i.e., an x-kernel interface. This is appropriate for the network, since *send* and *receive* is what one usually uses in this context (*u-send* and *u-recv* in Figure 3.1). This message-based interface is however too restrictive for the interface with the application. Consider for example an application sending (1) messages that need to be totally ordered, and (2) point-to-point reliable messages.

In Cactus, this would require to add some *type* information in the message in order for the protocol to be able to distinguish between the two types of messages.

In Appia, the interaction with the application and network is done by so-called asynchronous events (see Sect. 2.3.1). This is a much more flexible interface, allowing interactions more complex than just pushing/popping messages. Considering the above example, the application could trigger (1) *abcast* events and (2) *send* events. The only problem is that this works only from the application to the channel, but not the opposite. In the other direction, i.e., from the channel to the application, Appia does not provide any solution: the interface has to be implemented in an *ad-hoc* way. In our composition, *adeliver* has been implemented using a producer-consumer queue.

To summarize, Appia's application interface is more flexible, but it would be beneficial to have the same interface from the channels to the environment as from the environment to the channels.

3.3.3 Performance Comparison

We measured the performance of Cactus and Appia on the *Reliable Channel* protocol module between two processes, using the IBM SockPerf benchmark suite, version 1.2 [IBM00]. These benchmarks are designed to measure socket performance. In order to run these benchmarks, we implemented a special protocol module that interacts with the benchmarks: the SockPerf Proxy (see Fig. 3.4). Sockperf Proxy interacts with Reliable Channel to send/receive messages. The Reliable Channel protocol module is the same as in Section 3.2.

The hardware used for the measurements was (1) a 100 Base-TX Ethernet, with no third-party traffic, (2) two PCs running Red Hat Linux 7.2 (kernel version 2.4.18-19). The PCs have a Pentium III 766Mhz processor and 128 MB of RAM. The Java Virtual Machine was Sun's JDK 1.4.0.

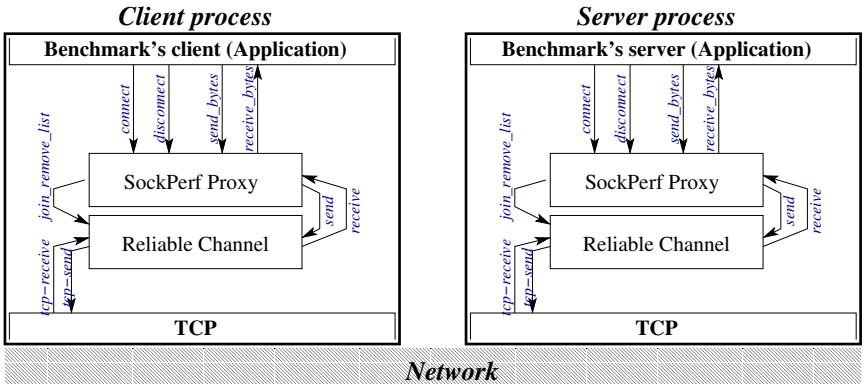


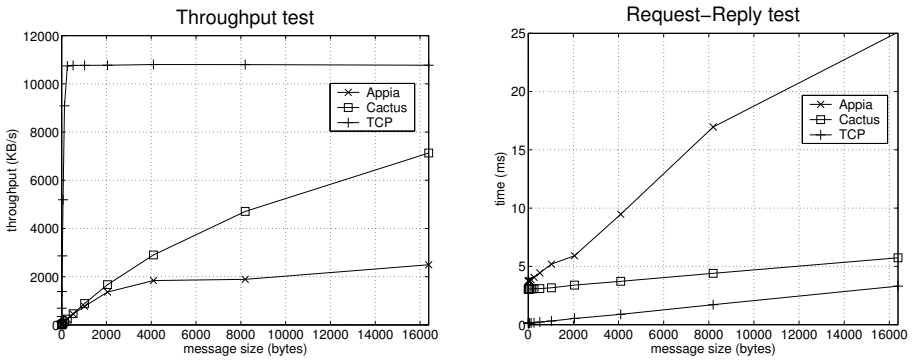
Figure 3.4: Benchmark configuration

3.3.3.1 Throughput

The first benchmark measures throughput with respect to message size. The benchmark does the following. First, the client connects to the server using event *connect* (Fig. 3.4), which causes a *join_remove_list* event to be triggered. This opens a socket at the TCP level. Once the socket is open, the client sends messages of a given size using event *send_bytes*. The handling of *send_bytes* results in triggering of *send*, which in turn triggers *tcp-send*. The client sends messages at maximum speed, and the throughput is measured.

In our Cactus implementation, as discussed in Section 3.2.2.3, the client benchmark's thread executes all the code in the composite protocol, including the *tcp-send* call on the socket. The TCP flow control blocks the calling thread if the sending buffer is full. In Appia, the client thread just inserts events *send_bytes* in the scheduler queue: execution of the event is carried by another thread. Because of the absence of flow control in Appia, our preliminary tests caused a memory overflow (overflow of the scheduler event queue). To prevent this, the designers of Appia added a preliminary flow control mechanism in the system, by bounding the size of the scheduler event queue: a caller wanting to add an event to the queue is blocked if the queue is full.

We have also measured the throughput directly using TCP sockets. The results are shown in Figure 3.5(a). The first observation is that the throughput grows linearly approximately up to messages of 2000 bytes. The growth slows down as we get closer to the network bandwidth limit (12.5 MB/s). This is not surprising: for large messages (e.g., 16384 bytes) the overhead of the framework becomes small compared to the network transmission time. Figure 3.5(a) shows also that for large messages the throughput obtained with Cactus is four times the throughput obtained with Appia. We come back to this issue later. The measurements also show that the overhead of both Cactus and Appia with respect to TCP is particularly significant for small messages.



(a) Throughput (b) Request-reply

Figure 3.5: Performance comparison

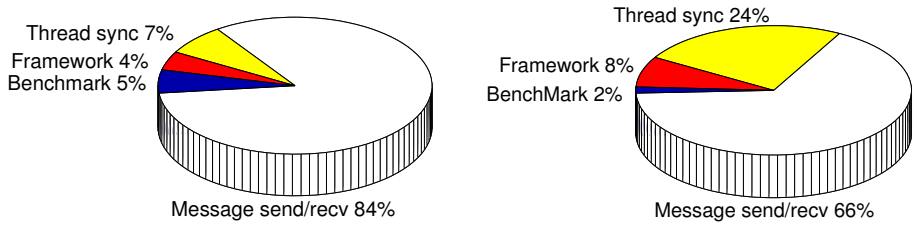
3.3.3.2 Request-Reply

The Request-Reply benchmark measures the round-trip time of a message. The client first connects to the server in the same way as it does for the throughput benchmark. Once the socket is open, the client sends a message to the server and waits. The server receives the message and immediately resends it to the client. The time elapsed at the client between the message sending (*send_bytes*) and reception (*receive_bytes*) is measured (averaged over several request-reply interactions).

We have also performed the same measurements directly using TCP. The results are shown in Figure 3.5(b), where we can see that Cactus performs again better than Appia: the factor varies approximately from 2 for short messages to 5 for large messages. Compared to TCP, the overhead of both Cactus and Appia is again quite significant, especially for small messages.

3.3.3.3 Execution Profiling

In order to understand the results of our experiments, we have used a profiler on the client side for the Request-Reply benchmark with 16384 byte long messages. We have classified the methods executed into five groups: (1) waiting on the socket, (2) benchmark-related methods, (3) framework-related methods (including protocol execution), (4) thread synchronization methods, and (5) message transmission and marshaling methods.



(a) Appia (b) Cactus

Figure 3.6: Profiling of the Request-Reply benchmark with messages of 16384 bytes (client side).

The results are depicted in Figure 3.6. The time spent waiting on the socket does not appear, since it is not relevant to locate bottlenecks. For both frameworks, we see that most of the time is spent treating messages (84% in Appia, 66% in Cactus).[6] The percentage is particularly high for the case of Appia. We believe that the reason is because Appia implements a library (class *ObjectsMessage*) to serialize objects, which itself uses standard Java marshaling. In Cactus, Java marshaling is used without any intermediate library.

[6] A similar result was obtained when profiling the execution on the server side.

In any case, the results show that message marshaling represents the bottleneck of both Cactus and Appia, and thus probably explains the results that appear in Figure 3.5. Serialization needs certainly to be done in a more efficient way. Standard Java class serialization is too heavy-weight for simple data types such as *byte[]* (used by Java sockets).

3.4 Proposals for Better Frameworks

After having analyzed the strengths and weaknesses of Appia and Cactus, we discuss in this section the most interesting features of the two frameworks, and propose improvements that could lead to better frameworks.

3.4.1 Composition Model

As we have seen in Sect. 3.3.2, the best and most general composition model is non-hierarchical, allowing protocol modules to directly communicate with each other. More restrictive compositions, e.g., the Appia channel, enforce an order that is sometimes not the right one.

3.4.2 Interaction Model

Point-To-Point Events. Our experience in building the Atomic Broadcast composition shows that the interaction models proposed by Appia and Cactus do not fit well the situation encountered most often. Indeed, the most frequent situation was a protocol module triggering an event, and one single protocol module handling it. We call such events *point-to-point events*. As discussed in Sect. 3.3.2, Appia's channels are not very well adapted for point-to-point events.

We propose an additional way to compose protocol modules, in which there is a point-to-point binding between the event triggered in one protocol and the incoming event in the other protocol module. It is a simplified version of the Cactus event binding scheme, where only one handler is bound to an event type. The Appia correction check could easily be extended to include this new binding rule.

Connectors. In Section 3.3.2.2, we have described the event multiplexing problem. With point-to-point events, the multiplexing problem can be solved with a special module called *connector*, which could be provided by the framework. A connector would allow a finer event routing. This is shown in Figure 3.7, which illustrates one particular type of connector called *multiplexor-demultiplexor*. If module m (respt. m') in Figure 3.7 triggers event $in1$ (respt. $in2$) at some process p, the multiplexor at the remote process q will trigger $out1$ (respt. $out2$). Note that this connector is different from the Cactus *demux* function (see Sect. 2.3.2), which deals with sessions, while the connector deals with micro-protocols. Cactus' coordinated event raising (Sect. 2.3.2) could easily be implemented as a connector.

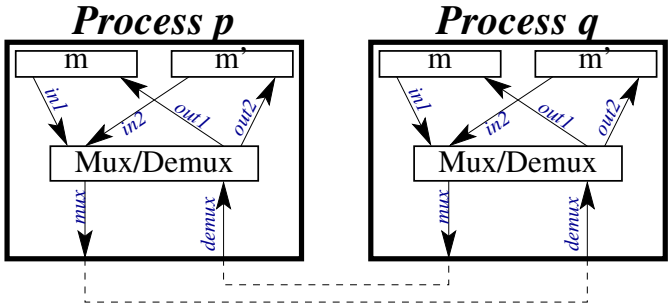


Figure 3.7: Multiplexor-demultiplexor connector.

3.4.3 Concurrency Model

To be general, the framework should allow multiple concurrency models in order to easily adapt to a changing environment (e.g., from mono-processor systems to multiprocessor systems). One of these models could be the Appia single-threaded model, which does not waste time in context switches. Another — more concurrent model — could be the *"thread per protocol module"* model (as in SDL, see Sect. 2.3.4), in which each protocol module has (1) its own thread, and (2) a (FIFO) queue of events, waiting for the protocol module thread to handle them. All the concurrency models should at least provide the following guarantees:

- *Default Synchronization:* Protocol programmers should be freed from the burden of expressing synchronization. This implies, for instance, that atomic handler execution should be guaranteed by default.
- *FIFO Event Handling:* FIFO order event handling is desirable in concurrency models in which the property makes sense. This includes the single-threaded model, and the "thread per protocol module" model.
- *Transparency:* Changing the concurrency model should be transparent to protocol modules.

In addition to these preliminary proposals, we will present more elaborate concurrency models in Chapter 4, which is devoted to the concurrency model.

3.4.4 Interface with the Environment

The interface with the environment can be divided into two issues: application/network interface, and interface to system resources.

In our opinion, a good interface from the application/network to the framework should be close to Appia (where the application/network can insert events into the framework). The interface from the framework to the application is trickier, since it depends on the application model. One solution for the framework is

to provide various interfaces, which fit most common application models. A good solution for interaction with the network is the one defined in JavaGroups [Ban02], where the application can choose a so-called *building block* from a repository (see Sect. 2.3.4). A building block is a predefined interaction scheme between application and composition (message queues, callbacks, etc.).

The access to system resources (typically timers) should be well integrated in the framework event model (which is the case with Appia and Cactus).

A crucial aspect for performance is serialization. We have noticed that the standard serialization provided by Java is not adapted to short messages. The framework should provide efficient serialization for primitive data types, and not just standard Java object marshaling (actually, a non-documented feature of Appia provides something similar).

3.5 Conclusion

In this chapter, we have analyzed Appia and Cactus, two frameworks for protocol composition. We have compared the two frameworks based on the experience gained in implementing an Atomic Broadcast composition. Our conclusion is that the optimal framework would inherit features from both frameworks: a composition and interaction model close to Cactus, several concurrency models that would include Appia's model, and an extended version of Appia's interface with the environment. It would also include new features like point-to-point events and a repository of connectors.

Chapter 4

Concurrency in Protocol Composition Frameworks

Recent protocol composition frameworks provide flexible interfaces, arrangements and communication patterns, and thus allow for finer-grained components, called protocol modules. Multi-threaded programming is the key to high performance in these frameworks. In this chapter, we investigate the concurrency models that such frameworks provide or should provide for programmers. Along with detailed discussions of the concurrency features of existing frameworks, we propose a set of features that can be offered to enhance the way frameworks manage concurrency.

4.1 Introduction

For several decades, distributed protocols have been structured as a set of collaborating components with more or less well-defined interfaces. Recent protocol composition frameworks have highly flexible interfaces, arrangements and communication patterns, that permit the use of finer-grain components, called *protocol modules*. The basic promise that modular protocol composition frameworks make is a full separation of concerns between the programming of protocol modules and their composition, to the extent that these two tasks can be carried out by different people with minimal interaction (see Chapter 1).

Multiprocessor machines are commonly used as servers, and are increasingly present on desktops, as well. Multi-threaded programming, which is extensively used in protocol composition frameworks, can take advantage of systems with multiple processors.

Contribution. Multi-threaded programming is significantly harder than single-threaded programming. Hence it is worthwhile investigating what support is available for programmers of distributed applications. This chapter concentrates on what support protocol composition frameworks provide to programmers. We survey the features of existing frameworks for multi-threaded programming, and ex-

tensively discuss the usefulness of each of the features. We focus on features allowing communication within a given process: in particular, concurrency issues that arise within a process. Communication between processes is much slower and often unreliable, and concurrency issues require different solutions, therefore it falls out of the scope of this chapter. We then propose a set of features to improve the way frameworks use concurrency. These novel features can be offered without significant changes in programs, and with a negligible performance hit. Among other things, we propose the following features: (1) sets of single-threaded protocol modules that can coexist with multi-threaded protocol modules, thus taking the best of two worlds; (2) non-overlapping execution of protocol modules involved in a chain of asynchronous communication, to avoid inconsistencies; and (3) providing ordering guarantees for asynchronous communication among protocol modules, including first-in-first-out (FIFO) and causal order, and an extension to causal order. To our knowledge, our definition for this particular extension of causal order is the simplest so far.

Perspectives for Framework Comparison. In Chapter 2, we defined four perspectives for framework comparison. One of them is the concurrency model: it defines whether and how concurrency is allowed in the framework. The present chapter focuses on the concurrency model of protocol composition frameworks with more details and more frameworks than Chapter 3.

Structure. Section 4.2 lists the protocol composition frameworks we investigated. Section 4.3 presents concurrency models of protocol composition frameworks. Section 4.4 compares the models and proposes a combination. Section 4.5 is concerned with overlapping executions in a protocol module. Section 4.6 defines and discusses ordering guarantees for the communication between protocol modules and their implementation. Finally, Section 4.7 concludes the chapter.

4.2 Protocol Composition Frameworks Considered

The following list contains the protocol composition frameworks considered in this chapter. We concentrated on frameworks that are still in use. Most of these frameworks are described in Sect. 2.3.

- Appia.
- Samoa.
- Eva.
- Neko.
- JavaGroups.
- Cactus (both C and Java versions).
- Fortika.

- JGroup [MDB01].

As Cactus has a two-level hierarchy of components (see Sect. 2.3.2), we often present Cactus in this chapter as two frameworks: *Cactus/μp* and *Cactus/cp* refer to how things work at the fine-grain and the coarse-grain levels, respectively.[1] Additionally, the concurrency features of the C and Java version of Cactus/μp are rather different. We will refer to them as Cactus/μp/C and Cactus/μp/J whenever the distinction is necessary.

Fortika is the group communication toolkit, developed within this thesis, that serves as a prototyping testbed for Part II (see Chapter 9). Fortika is relevant in this chapter because it follows a set of conventions and interfaces allowing us to write framework independent code (see Sect. 9.2). These conventions are indeed a synthesis of the features of three frameworks. In the rest of the chapter, when we cite Fortika, we refer to these conventions.

The JGroup framework is included because it is most similar to generic Java component frameworks: components usually hold references to other components and use method calls to communicate.

4.3 Concurrency Models

As the consequences of multiple threads running concurrently are difficult to foresee, one needs to coordinate how threads access shared resources, such as the internal state of protocol modules, or the shared state of multiple protocol modules. Frameworks differ in how they solve this problem (see Sect. 2.3). Some allow only a single thread for running protocol modules, and others are more permissive. In this section, we describe the characteristics of each group of frameworks.

Multi-threaded Model. A lot of frameworks (Neko, JavaGroups, Cactus/μp, Cactus/cp, Eva, and JGroup) permit the use of multiple threads. These frameworks provide special operations, protocol module skeletons, or protocol modules that launch new threads, or allow the protocol module programmer to use standard features of the programming language to launch new threads (Thread class in Java, POSIX threads in C, etc.).

These frameworks do not offer extensive support for restricting multi-threaded behavior; protocol module programmers and composers have to manage concurrency on their own.

Cactus/μp is unique among the frameworks surveyed in that the protocol programmer must specify how an event should be handled at the moment of triggering it: the event can be handled (a) by the triggering thread, (b) by a new thread, or (c) by a thread from a thread pool.

[1] μp stands for *microprotocol* and cp stands for *composite protocol*, the Cactus terms for components at the two levels.

Cactus/μp/C provides mechanisms to guarantee that the execution of a handler is not interrupted by the execution of another handler in the same higher-level protocol module, even in the presence of concurrency. In contrast, Cactus/μp/J does not restrict concurrency at all. We will return to Cactus/μp/C in Section 4.4.2.

Single-threaded Model. Two frameworks, Appia and Fortika (when used with Appia), have a dedicated thread that executes protocol modules. No other thread is allowed to execute protocol modules. Hence no concurrency is possible inside the composition.[2] The underlying philosophy is that multi-threading introduces a lot of complexity, and the possible gains in performance and readability are not worth this additional complexity.

The single-threaded model requires that protocol modules follow a set of strict rules:

- *No New Thread Launched:* Protocol Modules cannot start new threads or own private threads.
- *Non-Blocking Handlers:* Handling an event should not take a long time, and should never block the dedicated thread waiting for some condition. Otherwise, the handling of subsequent events may be blocked indefinitely, and the whole composition may have problems of liveness.
- *No External Interaction:* Protocol Modules should not interact with the outside world. This includes using the network, or peripherals. The reason is that such interactions may block, or may call the code of the protocol module asynchronously, thus introducing concurrency.

The rules can be summarized the following way: (1) protocol modules only trigger events while handling an event; (2) handling an event always finishes within a short time. Protocol Modules that follow these rules are called *reactive* protocol modules (they only react to their environment), and other protocol modules are called *active* protocol modules (e.g., they can launch new threads). The single-threaded model thus requires that all protocol modules be reactive. In contrast, both active and reactive protocol modules are allowed in the multi-threaded model.

Of course, frameworks that follow the single-threaded model also need functionality that involves external interactions. As reactive protocol modules cannot implement such functionality, external interactions are implemented by code that is not part of protocol modules, and are therefore *outside* the composition. The thread that executes protocol modules may communicate with a thread outside the composition using queues, for instance.

Finally, note that using a dedicated thread for executing the composition is not the only possibility to implement the single-threaded model. Another possibility is

[2] In Sect. 2.1, the term *composition* is defined as the set of all protocol modules and their interconnections within a process.

embedding the composition in a monitor that guarantees that only one thread can access protocol modules at any given time (this thread is not necessarily the same thread all the time). Fortika (when used with Cactus) uses this solution.

Model with Transparent Concurrency. The Samoa framework is unique in its approach to concurrency issues. It features a scheduler capable of allowing multiple threads to execute handlers concurrently, but, unlike in the multi-threaded model, this concurrency is *transparent* to the protocol modules: all protocol modules are reactive, just like protocol modules in the single-threaded model.

We have just seen that frameworks implementing the single-threaded model do not allow concurrency inside the composition. Therefore, if the environment triggers several events into the composition at the same time (e.g., several messages from the network), such frameworks handle these events one after the other. In contrast, Samoa's scheduler allows such events to progress into the composition unless they conflict (see [WRS04] for the exact definition of conflicts). In the best case, the events produced by external interactions can be handled concurrently, and otherwise, some events are blocked while other events are handled. The scheduler enforces an isolation property similar to the isolation property of transactions in databases. The kind of scheduler used is called a pessimistic (i.e., rollback-free) scheduler in that context.

To support the detection of conflicts, the composer has to provide additional information about the protocol modules that may or may not be executed for each type of external interaction. Such information is not necessary in either the single- or multi-threaded model.

Overall, this new approach is promising, but there remain open questions. One is how much concurrency can be introduced into the composition. For example, given compositions where most of external interactions execute most of protocol modules, the scheduler will hardly allow more than one event to progress concurrently. Other compositions probably fare better. Further research, involving measurements, is needed to answer this question.

4.4 Improvements for Existing Concurrency Models

In this section, we present the tradeoffs between single- and multi-threaded frameworks, and propose a combination. We then contrast this combination with the model with transparent concurrency, which can also be seen as a combination of the single- and multi-threaded models.

4.4.1 Drawbacks of Existing Concurrency Models

The Single-threaded Model Is Too Restrictive. Multi-threading is useful in a variety of scenarios. In particular, multi-threading may simplify the protocol code;

Table 4.1: Comparing the single- and multi-threaded models of concurrency and their combination.

Concurrency model	single-threaded	multi-threaded with reactive islands	multi-threaded
Protocol Modules	reactive, simple code		reactive or active, complex code
Active code	outside the composition	inside the composition	
Composition	simple		complicated

it is necessary for certain tasks, e.g., interfacing to system libraries that are multi-threaded; and it is useful to improve the performance of a service with slow operations. However, as *all* protocol modules are required to be reactive in the single-threaded model, multi-threading is forbidden within protocols. As a result, programmers are forced to solve important problems (that are inherently concurrent) or implement solutions efficiently *outside* the composition (e.g., within libraries).

The Multi-threaded Model Is Too Permissive. The multi-threaded model, in contrast to the single-threaded model, does not restrict the concurrency of protocol modules. This means that frameworks following this model provide few or no facilities for protocol programmers and composers to solve problems of concurrency, beyond the generic facilities offered by the programming language (locks, semaphores, monitors, etc.).

From the point of view of the protocol programmers, while many tasks are easy to implement in the absence of concurrency, they require complicated solutions if multiple threads are involved. For instance, the order of events can be reversed, and thus the protocol modules need to keep event order by themselves (e.g., by using sequence numbers).

From the point of view of the composers, concurrency issues (such as protecting the states of protocol modules against concurrent changes or avoiding deadlocks) cannot always be solved by the protocol programmers on their own: the composer must be involved. In order to do this, the composer needs to have deep knowledge of the composed protocol modules that includes details such as the cumulative state of the protocol modules to protect, or the handlers in which new threads are launched. Having to account for all these details greatly complicates the task of the composer. In contrast, the composer can ignore concurrency issues in the single-threaded model.

As stated in Sect. 3.4.3, frameworks should provide facilities to to restrict concurrency, which makes protocol programming and protocol composition easier. Frameworks following the multi-threaded model provide few such facilities.

4.4.2 Islands of Reactive Protocol Modules: the Best of Two Worlds

One can combine the advantages of the single- and multi-threaded models in the following way: let the programmer and the composer define sets of protocol mod-

ules that will be executed by a single thread at a time. We call such sets *islands*. Within islands, protocol programmers can write simpler code by taking advantage of the kind of guarantees offered in the single-threaded model: if all its protocol modules are reactive, the island itself is reactive as well. However, outside the islands, programmers are free to use multi-threading without restrictions, just as in the multi-threaded model, and thus implement any required functionality as protocol modules, within the composition. Table 4.1 summarizes the main characteristics of the single- and multi-threaded concurrency models (left and right columns, respectively) and points out how the multi-threaded model with reactive islands (center column) combines the advantages of both: protocol code and the task of composition is usually simpler, yet multi-threaded code can reside inside the composition.

Out of all the frameworks, only Cactus/μp/C follows a model similar to the multi-threaded model with reactive islands. This means that the framework allows active protocol modules, but its event scheduler provides guarantees for certain sets of protocol modules if they consist of reactive protocol modules only. The sets are the higher-level protocol modules of Cactus. The difference is that the boundaries of these sets cannot be chosen arbitrarily, as protocol modules and higher-level protocol modules work very differently in Cactus. Moreover, protocol programmers should not use certain features, e.g., triggering events in a new thread, in reactive islands.

Reactive Islands and Their Execution. We next elaborate on what we mean by reactive islands, and how they interface to other protocol modules.

The key property of islands is the following: if each of the protocol modules of an island is reactive, then the whole island behaves like a reactive protocol module, as well, and we speak of a *reactive island*. This means that reactive islands never trigger events if they are not handling any event; and if handling an event starts, it always finishes, and does so within a short time. Note that the single-threaded model uses exactly one reactive island that includes all protocol modules, whereas the multi-threaded model uses no reactive islands, or at least, the framework is not aware of reactive islands.

There are at least two solutions for executing an island that consists of reactive protocol modules only, such that the island itself will be reactive:

- One solution uses a dedicated thread. It is similar to how single-threaded frameworks handle events. There are two queues. Incoming events are put in the first queue, called inbound queue. A dedicated thread then repeats the following: it reads an event from the inbound queue, handles the event and any events triggered towards a protocol module in the island while handling the event, in a recursive manner. Outgoing events are put in the second queue, called outbound queue, from which other threads will read the events and handle them.

Table 4.2: Comparing the single-threaded, multi-threaded and transparent concurrency models.

Concurrency model	single-threaded	transparent	multi-threaded
Protocol Modules	reactive, simpler code		reactive or active, complex code
Active code	outside the composition	outside the composition but concurrency in the composition	inside the composition
Composition	simple	complicated	

- The other solution is that the island forms a monitor. Any thread can enter to handle an event, but in mutual exclusion with other threads. Inside the monitor, the thread should handle the event and any events triggered towards a protocol module within the island, in a recursive manner, but not any outgoing event. It may handle outgoing events only after exiting the monitor.

Using Islands of Protocol Modules. What parts of the protocol stack or distributed application can and should be implemented as an island of reactive protocol modules? Which ones should be active (i.e., multi-threaded)?

Our performance results in Chapter 3 show that, in usual compositions, most of processor time is spent on the serialization and deserialization of events for transmission over the network. Moreover, if the framework implements a single-threaded model, (de)serialisation becomes a prominent performance bottleneck.

In multiprocessor platforms, serialization workload produced in a typical protocol composition is big enough to fully utilize all processors available at each node. Besides, parallelizing serialization is extremely easy since the (de)serialization of two different events is completely independent.

This leads us to the conclusion that most of the code of the distributed application can be implemented as a single reactive island of protocol modules, whereas serialization-related protocol modules should be executed by multiple threads so that the high workload they generate is distributed over all processors of the execution platform. Note that if the application has different performance characteristics (e.g., serialization is no more the bottleneck), it is often straightforward to shrink the reactive island's boundary to introduce concurrency, by adding active protocol modules or replacing some of the reactive protocol modules by active versions.

4.4.3 Comparing with Transparent Concurrency

Just like the multi-threaded model with reactive islands, the model with transparent concurrency can be considered as a combination of the single- and multi-threaded concurrency models. Table 4.2 points out how the model with transparent concurrency (center column) combines aspects of the single- and multi-threaded concurrency models (left and right columns, respectively): concurrency is possible within the composition, just like in the multi-threaded model, and the price is that the composer's task becomes difficult, as the composer has to provide rather detailed

information about possible executions. Other aspects are similar to the single-threaded model.

Let us now contrast the multi-threaded model with reactive islands and the model with transparent concurrency. With both models, (most) protocol modules are reactive. However, concurrency is introduced in a different way. The model with reactive islands allows active protocol modules, and the composer can compose these with reactive protocol modules. In contrast, the model with transparent concurrency puts all the burden of introducing concurrency on the composer, and the composer's task is much more difficult.

Note that the model with transparent concurrency still requires that external interactions are performed outside the composition. In other words, this model only aims at solving one problem: that of increasing performance on multiprocessor platforms. For this reason, we view the two approaches as complementary, and in fact, they could be combined. The combination is a multi-threaded model with reactive and active protocol modules, in which some reactive protocol modules form reactive islands and some others form islands that are executed by a scheduler offering transparent concurrency. The latter islands are like the entire composition in the model with transparent concurrency.

4.5 Avoiding Overlapping Execution of Handlers

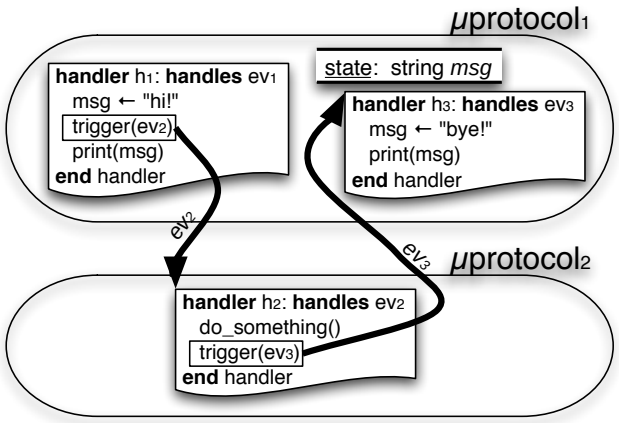


Figure 4.1: Example of possibly overlapping executions of handlers. What are the contents of msg when handler h_1 prints it?

This section discusses techniques to prevent the execution of a handler from starting before the handler that triggered the event is done with its execution. Let us illustrate this with an example. Consider an execution that involves handlers

h_1, h_2, and h_3. Handler h_1 triggers an event that is handled by handler h_2, and h_2 triggers another event handled by h_3. Finally, let h_1 and h_3 be handlers of the same protocol module $\mu protocol_1$. Figure 4.1 depicts this execution. Such chains of events, which that start and end at the same protocol module, may give rise to consistency problems. In the example, h_1 and h_3 are part of the same protocol module ($\mu protocol_1$), and thus they both modify the state of this protocol module (the string msg). The consistency problem is that the output depends on how the statements of h_1 and h_3 are executed: if the first line of h_3 executes before the last line of h_1, h_1 will print the wrong message ("bye!").

Note that the root of the problem is that the executions of handlers h_1 and h_3 may overlap in time. Note also that the protocol programmer cannot use standard mechanisms (*synchronized* reserved word in Java, *mutex* variables in C) to guard against the consistency problems. The reason is that such mechanisms only protect against modifications by *different* threads, and h_1 and h_3 might indeed be executed by the *same* thread (e.g., if triggering is implemented with a function call).

This section describes and discusses possible solutions to the problems caused by overlapping executions of handlers.

4.5.1 Anticipating Consistency Problems

The first possible solution is requiring the protocol programmer to anticipate consistency problems (or, at least, to anticipate consistency problems that may occur when a single thread executes several handlers of the protocol module such that they overlap). This would require the programmer to know the details of the composition (e.g., $\mu protocol_1$ and $\mu protocol_2$ in Fig. 4.1). This goes against the main property of protocol composition frameworks: protocol modules should be written in (relative) isolation and composed in a (relatively) unrestricted manner. Hence we do not discuss this solution any further.

4.5.2 Non-Overlapping Handler Executions

A better solution is that the framework implements a mechanism that explicitly prevents overlapping handler execution, requiring that a handler h finishes before the handling of any event triggered by h. If such a mechanism is present, the consistency problem described above cannot occur: the execution of handlers h_1 and h_3 in Figure 4.1 will never overlap.

We next present two possible mechanisms to avoid that the execution of handlers overlaps:

- *Scheduler with an Event Queue.* Non-overlapping execution of handlers can be implemented by putting triggered events into a queue, and handling the events in this queue only after the triggering handler finishes. Appia and Cactus/μp/C use this implementation. The advantage of this solution is that it is implemented in the runtime system; protocol programmers do not have to worry about ensuring non-overlapping executions when writing their code.

- *Conventions for Writing Handlers.* Non-overlapping execution of handlers can also be ensured by convention. Triggering an event causes handling the event immediately: the method implementing the handler is called directly from the triggering handler. Normally, this would result in overlapping handler executions. This can be avoided if protocol programmers follow rules that ensure that the resulting execution is equivalent to a non-overlapping execution. Fortika protocol modules and most of the Neko protocol modules follow such rules.

  We now present two examples of such rules. The first rule requires that triggering events be the last actions of a handler: if events are generated before the end of the handler, they must be stored in local variables and triggered at the end of the handler. The second rule states that handlers copy all the data necessary for their execution into local variables before any events are triggered. If this rule is followed, the execution of a handler is not affected if any of the triggered events causes the execution of a handler of the same protocol module.

  The example in Fig. 4.1 clearly does not follow either of these rules. To make the protocol modules follow the rules, one needs to move trigger(ev_2) in h_1 to the end of the handler.

Discussion. Conventions for writing protocol modules restrict protocol programmers. However, the advantage is that a single Java method call (or C function call) implements triggering and handling an event. This yields good performance, as synchronous calls are the primary means of interaction between different parts of the code in all popular programming languages and are thus significantly faster than any other communication mechanism. Moreover, method/function calls allow the compiler to check the types of the data transmitted in an event, whereas other communication mechanisms are not as type-safe.

To summarize, conventions for writing protocol modules offer better performance and type-safety, but they are slightly inconvenient for protocol programmers. Our conclusion is that none of the two solutions is conclusively better than the other, and thus we consider that either solution yields good protocol composition frameworks.

4.6 Ordering Events

A protocol module's effect on other protocol modules and the environment is not just determined by the set of outgoing events that the protocol module triggers. Often, the order in which the outgoing events are triggered is important as well. An everyday example is requiring first-in-first-out (FIFO) guarantees from a communication channel, such as a TCP connection. Implementing protocols such as HTTP on top of TCP is much easier than it would be to implement it on top of

a reliable datagram service without FIFO guarantees. Note that providing FIFO guarantees between the two communicating processes is not enough. FIFO ordering is needed within each process, between the code that implements HTTP and the code that implements TCP, as well.

Table 4.3: Conceptual mapping between communication in message passing systems and protocol composition frameworks.

Systems	message passing distributed systems	composition frameworks
Comm. entities	processes	protocol modules
Unit of comm.	messages	events
Start of comm.	sending	triggering
End of comm.	receiving	start of handling
Comm. steps	events	actions

A variety of ordering guarantees have been introduced in the field of message passing distributed systems, starting with early papers [Lam78]. Events are an asynchronous form of communication, just like messages in distributed systems. Protocol modules communicate by events and may be executed in parallel, by different threads; they are like processes in distributed systems that communicate by messages and execute in parallel. The two kinds of systems are compared in Table 4.3. This section investigates whether ordering guarantees proposed in message-passing distributed systems are relevant for protocol composition frameworks.

4.6.1 Feasibility of Ordering

FIFO order ensures the following: if two events are triggered from the same protocol module to the same handling protocol module, then the order of handling is the order of triggering. We also consider extensions of FIFO order, such as causal order, well-known in distributed systems [BJ87], as well as an extension to causal order.

These orderings are motivated, defined and discussed later, in Section 4.6.2. We first investigate how protocol composition frameworks should support event ordering .

Ordering Events in the Presence of Active Protocol Modules. Protocol composition frameworks should not guarantee order between events if active protocol modules are involved. The reason is that the framework must supervise the actual order in which protocol modules trigger and handle events. This supervision task requires that the threads involved are synchronized frequently, otherwise it is hard to tell which of two concurrent actions happened first. However, a major motivation for using active protocol modules is to fully exploit multiprocessor systems,

and this is only achieved if threads are synchronized infrequently. In other words, ordering events if active protocol modules are involved defeats the reason for using active protocol modules.

Ordering Events within a Reactive Island of Protocol Modules. The framework should offer support for keeping order to a certain extent: events within a reactive island of protocol modules should be subject to ordering.

The reason is that keeping even the most complicated kind of ordering (discussed and motivated in Section 4.6.2) is cheap in this context: no thread synchronization is involved, as all events triggered or handled in a reactive island are processed by a single thread at any time. We present algorithms for ordering within reactive islands in Section 4.6.3.

4.6.2 Definitions of Ordering

This section reviews the orderings we consider, provide formal definitions, and present related research results, also from other contexts.

We introduce the following notation: $\mathrm{trigger}(e)$ denotes the action of triggering some event e, $\mathrm{handler}(e)$ the protocol module that handles e, and $\mathrm{handle}(e)$ the action of starting the handling of event e, i.e., the point in time when execution of $\mathrm{handler}(e)$ starts.

We also introduce precedence relations on actions. Actions are either the triggering or the handling of an event. Definition 1 is a precedence relation that orders actions that take place on the same protocol module.

Definition 1 (Local Precedence) *Consider two actions A and B that take place on the same protocol module. A locally precedes B ($A \rightarrow_l B$) if (1) A and B are executed by the same thread and A is executed first, or (2) A and B are ordered by a synchronization primitive of the programming language and A is executed first. Local precedence is the transitive closure of these relations.*

For example, A and B are ordered by a synchronization primitive of the programming language if A and B are part of synchronized blocks of a Java program that synchronizes on the same object, even if they are executed by different threads. Note that not all actions of a protocol module are ordered by local precedence. For example, actions executed by different threads on a protocol module that does not use synchronization facilities are not ordered. FIFO order is defined in terms of local precedence:

Definition 2 (FIFO Order) *Let e and e' be two events such that* $\mathrm{handler}(e) = \mathrm{handler}(e')$. *If* $\mathrm{trigger}(e) \rightarrow_l \mathrm{trigger}(e')$ *then* $\mathrm{handle}(e) \rightarrow_l \mathrm{handle}(e')$.

Informally, FIFO order means the following: if two events are triggered by the same protocol module to the same protocol module, then the order of handling must

be the order of triggering. Other orderings involve a different kind of precedence:[3]

Definition 3 (Causal Precedence) *Consider two actions A and B. A causally precedes B ($A \rightarrow B$) if*

(a) $A \rightarrow_l B$,

(b) $A = \mathrm{trigger}(e)$ and $B = \mathrm{handle}(e)$ for the same event e, or

(c) there exists an action C such that $A \rightarrow C$ and $C \rightarrow B$ (transitive closure).

Additionally, we say that actions A and B are concurrent ($A \parallel B$) if $A \not\rightarrow B$ and $B \not\rightarrow A$.

Causal precedence allows us to define causal order:

Definition 4 (Causal Order) *Let e and e' be two events such that* $\mathrm{handler}(e) = \mathrm{handler}(e')$. *If* $\mathrm{trigger}(e) \rightarrow \mathrm{trigger}(e')$ *then* $\mathrm{handle}(e) \rightarrow_l \mathrm{handle}(e')$.

Causal order is a generalization of FIFO order. FIFO order concerns events triggered within *the same* protocol module; in contrast, causal order concerns events triggered at *different* protocol modules (as long as they are related by causal precedence).

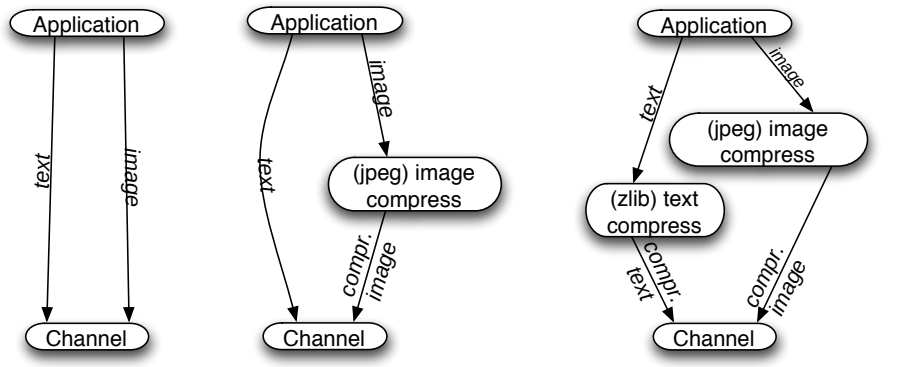
Extended Causal Order. We now present an example that illustrates the utility of FIFO and causal order in the context of protocol composition frameworks. The example also motivates the introduction of a new notion of order: extended causal order.

Consider the two protocol modules illustrated in Figure 4.2(a). The application protocol module has to send a message with both text and image data, which are conveyed by two different events. The order is important: the text is sent before the image, hence a FIFO communication channel, implemented by module Channel, is used. Thus, the communication between the two protocol modules must be FIFO.

Now consider a variant of this scenario, shown in Figure 4.2(b). Again, the application protocol module triggers the text event before the image event. The difference is that the image data is now compressed by a third protocol module. In this variant, one needs causal order. With FIFO order, it is possible that module Channel handles the compressed image *before* the text. If this happens, the message would be garbled.

Finally, consider another variant of the scenario, shown in Figure 4.2(c). The difference is that the text is compressed by a fourth protocol module (the third protocol module cannot be reused, as image data and text are usually compressed with different algorithms). In this variant, neither FIFO nor causal order can guarantee that module Channel handles the compressed text first and the compressed image second. The reason is that the triggering of the events conveying the compressed

[3] The distinction between $\rightarrow_l$ and $\rightarrow$ can be removed. We have decided to keep it, though, because it clarifies the presentation.



(a) no compression (b) image compressed (c) both image and text compressed

Figure 4.2: Example to illustrate FIFO, causal and extended causal order.

image and the compressed text are *not* related by causal precedence (they are concurrent!). Hence the need to extend the notion of causal order.

Causal order, as defined above, only considers events whose triggerings are related by causal precedence. Although this is usually enough in distributed systems, protocol composition frameworks need a stronger notion of ordering that rather considers the very beginning of a chain of events. We propose a novel notion of order, which implies causal order (and therefore FIFO order):

Definition 5 (Extended Causal Order) *Let* e_t, e'_t, e_h *and* e'_h *be events such that*
(1) $\text{trigger}(e_t) \rightarrow \text{handle}(e_h)$,
(2) $\text{trigger}(e'_t) \rightarrow \text{handle}(e'_h)$, *and*
(3) $\text{handler}(e_h) = \text{handler}(e'_h)$ *(possibly* $\text{trigger}(e'_t) \not\rightarrow \text{handle}(e_h)$*).*
If $\text{trigger}(e_t) \rightarrow_l \text{trigger}(e'_t)$ *then* $\text{handle}(e_h) \rightarrow_l \text{handle}(e'_h)$.

While causal order involves triggering events e_1 and e_2, and handling *the same* events e_1 and e_2, extended causal order involves the triggering of events e_1 and e_2, and handling of e'_1 and e'_2 that causally precede e_1 and e_2, respectively (which are not necessarily the same events).

Enforcing extended causal order in the example of Figure 4.2(c) solves the problem.

Related Work. Causal precedence corresponds to Lamport's precedence relation in message passing distributed systems [Lam78]. The difference is that in [Lam78], all events on a process are ordered, whereas our local precedence is a partial order.

Local and causal precedence are similar to relations introduced in the context of distributed object environments [DFS00]. The main difference is that our model is simpler: besides asynchronous communication, [DFS00] considers synchronous communication and read/write operations on shared variables, as well.

Our definition for causal order is analogous to causal order in message passing distributed systems [BJ87] and distributed object environments [DFS00].

[Yos01] defines extended causal order. Our definition greatly simplifies that definition.

Real-Time Order. Besides the order of events, the time of triggering and handling events may also be important, e.g., for real-time applications. We do not expect that the majority of applications of a protocol framework would be real-time, hence we do not think that the cost of automatic timestamping can be justified. Moreover, timestamping is a relatively expensive operation, as it often involves a system call.

Nevertheless, if protocol modules need timestamps they can be provided easily. Either protocol modules can explicitly put a timestamp into the events they trigger, or the composer can add timestamping protocol modules to the composition.[4]

4.6.3 Implementations of Ordering

In Section 4.5.2, we presented two scheduler implementations: one with direct method calls and another with event queues. The implementation with direct method calls ensures extended causal order. The implementation with event queues also ensures extended causal order if all newly triggered events are inserted at the front of the event queue in the order in which they were triggered.

All frameworks but Appia and Cactus/μp/C use a scheduler with method calls, and thus keep extended causal order in executions that involve a single thread.

Cactus/μp/C uses event queues. Recall that Cactus has one trigger operation that has the event handled in the triggering thread, and another that has the event handled in a different thread. If all trigger operations are of the first kind (i.e., the execution involves a single thread), Cactus/μp/C provides extended causal order. Otherwise, it provides only casual order. The reason is that newly triggered events are inserted at the end (rather than the front) of the event queue. No guarantees are provided beyond the boundary of the higher-level protocol module, though.

Appia does not keep causal order or extended causal order. To explain why, we describe how its scheduler works. Appia organizes protocol modules in stacks, and events are associated with a direction: up or down, depending on the position of the handling protocol module in the stack with respect to the position of the triggering protocol module. When an event going up is handled, events triggered by the handler are placed at the front of the event queue if they go up, and at the end of the event queue if they go down. This breaks causal order for the scenario shown in Figure 4.2(b) if we assume that (1) the scenario is triggered by an event going down (not shown in the figure), (2) the channel protocol module is at the top,[5]

[4] Real-time scheduling of handler executions, if needed, is a more complex problem.

[5] In order to keep a realistic example, assume *channel* is actually a timestamping protocol that has to be high in the stack for compositional reasons.

(3) the application protocol module is in the middle, and (4) the image compress protocol module is at the bottom.

It is unclear why Appia has been designed this way. Note that changing Appia to ensure causal and extended causal order would be rather easy. The same holds for Cactus/μp/C and extended causal order.

4.7 Conclusion

In this chapter, we focused on how protocol composition frameworks deal with concurrency. Multi-threading is often useful in this context. We investigated what support for multi-threaded programming such frameworks provide and should provide for programmers.

Along with a survey and detailed discussions of the features of existing frameworks, we proposed a set of features that can be easily offered, and that have a negligible performance impact. This includes (1) islands of protocol modules with reactive behavior that can coexist with active protocol modules, thus taking the best of two worlds; (2) non-overlapping execution of protocol modules involved in a chain of events, to avoid inconsistencies; and (3) ordering guarantees for handler invocation, including causal order and extended causal order. To the best of our knowledge, our definition of extended causal order is simpler than existing definitions.

Chapter 5

The Header-Driven Model

Most state-of-the-art protocol composition frameworks use events at the core of their interaction model because they capture well the reactive nature of distributed algorithms. Besides, they allow a clean decoupling between different protocol modules, making their interface clearer. As a result, events may seem to be the only unquestionable concept that nearly all protocol composition frameworks use. However, we question their appropriateness in this chapter, enumerating several important drawbacks. Instead, we propose a new interaction scheme that shifts the spotlight from events to messages headers. We show how the new model overcomes the drawbacks of events.

5.1 Introduction

In Chapter 2, we defined four perspectives for framework description and comparison. The focus of the present chapter is on two of these perspectives: the composition and interaction models. Issues related to the concurrency model and the interaction with the environment (which were studied in Chapters 3 and 4) are orthogonal to the problems discussed and the solutions presented in this chapter.

The common denominator of state-of-the-art protocol composition frameworks is (1) the event-driven interaction model (see Section 3.3.1.1), by which protocol modules notify their surrounding modules about things that occur, and (2) the trend to graph-based composition models (see Section 1.1), where a protocol module can directly interact with several neighboring modules.

Most real-life compositions are deployed in a symmetrical manner: all processes contain the same protocol modules, and they are interconnected in the same way. As discussed in Sect. 2.1.5, the *peer interaction* is a frequent pattern that takes place in virtually all symmetrical compositions. Roughly speaking, a peer interaction takes place when a protocol module wishes to interact with its peer protocol module at other processes. To do so, it uses a local protocol module (which can itself launch another peer interaction) that provides a lower-level service. Our claim is that current protocol composition frameworks do not properly handle peer

interactions. As a result, the fact that the composition is the same at every process is not exploited.

Although the event-driven interaction model is widely used by protocol composition frameworks, it is far from perfect. The way it handles the omnipresent peer interactions is (1) complex (the composer needs to do many unnecessary bindings), (2) obscure (the indirections introduced by events blur the presence of peer interactions in the protocol code), and (3) unsafe (misbindings may lead to runtime errors or erratic behavior). Instead, we propose a different way to look at inter-module interaction, dropping events and their bindings. The result, is a new interaction model: the header-driven model. This model (1) solves the compositional problems found in the event model, (2) simplifies inter-module dependencies, (3) concisely handles peer interactions and explicitly reveals their logical structure, and (4) provides better support for type-checking at compilation time, which avoids the runtime errors and erratic behavior that can occur in the event-driven model.

In short, the contribution of this chapter is a discussion about the inadequacy of the event-driven model, as well as the new header-driven model. Section 5.2 reviews the basic features we expect from any protocol composition framework in this chapter. Section 5.3 presents a typical event-driven interaction model and discusses its main drawbacks. Section 5.4 presents the novel header-driven model and explains how it overcomes the problems presented in Section 5.3. Section 5.5 makes a direct comparison using a tiny but representative example implemented in both models. Finally, section 5.6 concludes the chapter.

5.2 Assumptions on the Framework

Before getting to the main discussions of this chapter, we describe several basic features we assume from the protocol composition framework. These are general features present in state-of-the-art protocol composition frameworks. Both interaction models presented in Section 5.3 (event-driven) and Section 5.4 (header-driven) can be *plugged* into this general abstract framework.

5.2.1 Programming Language

One of the popular general-purpose programming languages (such as Java, C, etc.) can be used to implement the models discussed here. The proposed interaction model can be encoded as a tiny library. Nevertheless, there is a particular feature that would be beneficial if present (but is not required): the possibility to treat function/method calls as *first-class values*: it must be possible to decide whether to execute a function call immediately, or to store it in some data structure for future execution.

5.2.2 Composition and Interaction Models

Symmetric Graph-Based Compositions. We assume that the framework permits graph-based compositions. In particular, a setup with a lower-level protocol module l offering its services to *several* higher-level modules $h_1, \ldots, h_n$ should be possible. Ideally, each module h_i should not be aware that other modules are using l. This is not the case quite often in current frameworks. We come back to this issue later on.

On the other hand, we only consider symmetric compositions: those with the same composition at every process (see Sect. 2.1.5). Most implementations of distributed algorithms can easily be expressed with symmetric compositions. Peer interactions are a particularly frequent interaction pattern of symmetric compositions (see Sect. 2.1.5). In a peer interaction, a protocol module h intends to communicate with its a peer module h', but it does it indirectly: it uses some lower-level protocol module l that, together with its peer l', offers the service needed so that the communication between h and h' can take place.

Protocol Module Instances. We extend the terminology presented in Sect. 2.1 to make the important distinction between *protocol module* and *protocol module instance* (also called *protocol session* in the literature). The former is the set of handlers and the structure (type) of the private state, whereas the latter is a particular copy of the protocol module with a value for its state. Several instances of the same protocol module can coexist within the same process. These instances behave independently from each other, having their own state, although all share the same code.

5.2.3 Interface with the Environment

The details of the interaction between the application and the framework is not relevant for this chapter. On the other hand, we need access to the network subsystem through a standard networking library. The minimal quality of service expected from the network is unreliable point-to-point message transmission.

5.3 Shortcomings of the Event-Driven Model

A protocol module is usually designed to be idle until it receives some notification from surrounding modules, to which it reacts by executing some code that may notify other modules, and goes back to its idle state, waiting for other notifications. These notifications are indeed the *events that drive* the interaction model of most protocol composition frameworks.

As we have seen in the previous chapters, frameworks vary widely on how they define event-driven primitives, as well as the rules that govern their itinerary from the triggering to the handling module(s). This section is an attempt to simplify all those models by *factoring out* a set of key elements that fully characterize their

common essence. Our intention is to keep the discussion focused, rather than getting lost among this or that framework's unique features. Once the basic model is presented, we proceed to discuss the main drawbacks of the event-driven model.

5.3.1 An Abstract Event-Driven Model

Protocol modules are declared with reserved word protocol and consist of four parts:

- State declaration, using reserved word state, which consists of a list of variable names, along with their corresponding types.
- Event type declarations, using reserved word event. The declaration includes the name of the event type, and the arity and type of arguments conveyed within the event.
- Handler declarations, using reserved word handler. This includes the handler name, the formal parameters and the body of the handler. The body implements the protocol and typically changes the state and/or triggers events using reserved word trigger.
- Bindings from local handlers to events, using reserved word bind. A binding has two parameters, the first is a handler name, the second is the event type to which the handler is bound.

An simple example of a protocol module follows:

```
protocol counter is
  state
    int count
  event maxReached(int max)
  handler handleTick() is
    count ← count + 1
    if count > MAX then trigger maxReached(count)
  bind(handleTick, prot1.tick)
```

In this example, protocol module counter counts the number of times that protocol prot1 (omitted) triggers event tick. If the number of ticks has reached constant MAX, the handler triggers event maxReached containing the current count. Of course, a protocol module interested in being notified by event maxReached should bind a handler to it, in the same way as the depicted protocol binds its handler to event tick (declared within prot1).

There is no restriction as to how many handlers can be bound to the same event, or vice versa. When an event is triggered, all handlers bound to it should be executed. The way they are executed (sequentially, concurrently, etc.) differs form framework to framework and depends on its concurrency model. Therefore, this question is out of the scope of the chapter.

5.3.2 The Event Routing Problem

The event routing problem occurs within a process. Consider Fig. 5.1, where the high-level protocol module A_1 relies on the service provided by a local lower level protocol B. The intended behavior is that A_1 issues a request and waits for the reply from B. This is typically implemented by two event bindings, one going down (the request) an another going up (the response). Now, assume another high-level protocol A_2 uses the same low-level protocol B. If we use the same solution, there are two down-going bindings for the same handler at B. More importantly, there are two handlers, one at A_1 and the other at A_2 that will be executed when the up-going event is triggered. This setup is depicted in Fig. 5.1, left.

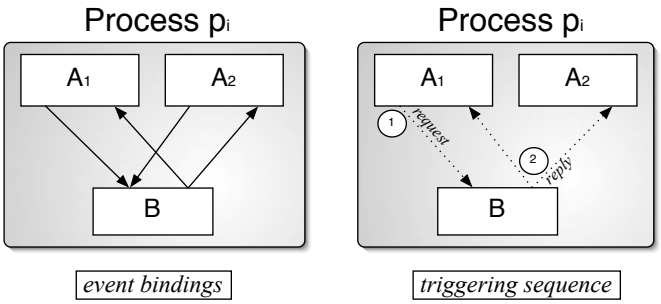


Figure 5.1: Example of the event routing problem: A_2 receives an unexpected event.

The problem arises (Fig. 5.1, right) when A_1 issues a request (triggers the down-going event). In this case, protocol B triggers the event carrying the response and *both* A_1 and A_2 will receive the response. While this may be desirable in some setting, it is not the expected behavior most of the time: only the module issuing the request should receive the response. From the modularity point of view, it is annoying that the designer of A_2 has to know, *a priori* that A_2 may be composed together with A_1, and therefore receive events not aimed at it. Indeed, both A_1 and A_2 should perceive the composition as though they were alone using B. Hence, we need to route B's response so that it only reaches the protocol module that expects it.

5.3.3 Ad-hoc Solutions to the Event Routing Problem

There are some ways to solve the event routing problem in the event-driven model, but they are inefficient with respect to modularity and performance. We now present the best solutions to reach a balance between modularity and performance:

- *Destination Check on Handler Invocation*. Additional data in the event's arguments can be included. Upon event reception, a check is included to

ensure that the event is targeted to this module, otherwise the event is discarded. This solution burdens the programmer of the protocol module and induces a performance overhead if the check is to be done for every triggered event.

- *Multiple Low-Level Modules*. Another way to cope with the problem is that every high-level protocol module uses a distinct instance of the lower-level protocol module. This way, no protocol module is shared, and up-going events are trivially routed to the right destination protocol module. The downside is that it does not scale, rapidly ending up with numerous instances of the same protocol module at lower levels of the composition. Besides, sometimes it is simply not possible to have more than one instance of a protocol module because it would violate the module's semantics. As an example, consider a protocol module that is to provide message transmission with FIFO order. If there are two instances of this protocol module, each one will provide FIFO order to the messages it receives, but there will be no order between messages received from different module instances.

- *Connectors (demultiplexors)*. This is the most acceptable solution and was introduced to solve a similar problem in Sect. 3.4.2 (also, a similar version is used by Cactus at the composite level). They are special modules that locally route events to the right destination module, without any intervention of normal protocol modules. There are several types of off-the-shelf connectors and the composer chooses from a repository. The *multiplexor-demultiplexor* connector presented in Sect. 3.4.2 solves the event routing problem. This approach, while neatly more elegant for modularity than the two previous solutions, still has major inconveniences.

  - The composer needs to get familiar with connectors, and has to plug them at the right place so that the composition works.
  - Connectors are extra modules added to the composition. They thus introduce a performance overhead, since messages have to traverse them.

Another solution is using events as call-backs: the request (down-going) event contains within its parameters the response event type to trigger. This would work fine, although there is a drawback: the composer can no more make explicit bindings for certain event types. This call-back technique is a preliminary version of the core idea behind our header-driven model.

5.3.4 Peer Interactions in the Event-Driven Model

As described in Sect. 5.2.2, a peer interaction takes place when a protocol module uses an indirect strategy to communicate with its peer modules at other processes. The module that starts the communication creates a message where it pushes the information intended for its peer module, then it triggers an event that conveys the

message to a local lower-level module. The message eventually arrives to the peer protocol module at another process, and the peer module pops the information from the message. Such units of pushed/popped information are called headers (see Section 2.1).

Figure 5.2, shows a simple composition with protocol modules R and U at one process and their corresponding peers R' and U' at another process. Assume U and U' provide an unreliable message transmission passing service (some messages may get lost), whilst R and R' are a reliable message transmission service implementation that relies on U and U'. The left half of the figure depicts with solid arrows the event bindings necessary to allow for peer interactions. Protocol module R starts a peer interaction by pushing a header (containing the information for R') into a message and triggering an event bound to the sending handler at U, say U.handleSend. U handles that event: among other things, it pushes its own header to the message and transmits it through the network. If the message is not lost, U' receives it and pops the header U pushed, necessary for proper operation of the protocol. Then, U' triggers an event bound to the receiving handler of R', say R'.handleReceive, so that the message finally reaches R'. Finally R' handles the event: it pops the header R pushed at the beginning. This example contains two peer interactions, one between U and U' and the other between R and R'.

The way peer interactions are implemented in the even-driven model has the following drawbacks:

- *Compositionally Suboptimal.* When R starts a peer interaction, it does know that R'.handleReceive should be executed at R'. R has this knowledge because R' is its peer, moreover, R and R' are usually two instances of the same protocol module. However, this a-priori knowledge cannot be reflected in R's code. Instead, R triggers an event and *hopes* that the composer will do the right bindings so that R'.handleReceive will be executed. The level of indirection introduced by these bindings prevents the programmer of R from encoding an information she already knows. This is compositionally suboptimal, since a fact known by the protocol programmer should be encoded by her, and not later. In the event-driven model, this has to be enforced at composition time.

  Furthermore, R'.handleReceive can be defined internally to R' (and its peer R) so that it does not appear in their interface. When R starts a peer interaction by triggering an event bound to some handler of U, its real intention is to have its peer, R', execute R'.handleReceive. Despite this, handler R'.handleReceive needs to be exposed in R and R''s interface, so that an event triggered by U (or U') can be bound to that handler at composition time. This is clearly not modular, since something semantically internal to a protocol module should not be present in its interface.

  The call-back mechanism mentioned at the end of Section 5.3.3 can alleviate this problem, since the event bound to handler R.handleReceive (or

R'.handleReceive) can be passed to U as a parameter when the down-going event is triggered. However, there are still two dependencies between R and U (conversely R' and U'), which need two bindings: one for events going down and the other for events going up.

- *Event Routing Problems*. As we have seen for the general case, if U is used by more than one module, we run into the event routing problem.

- *Poor Type Information for Messages*. Messages are structured as (LIFO) header stacks. The nature of the data contained in message headers can be very heterogeneous. So frameworks following the event-driven model usually give the imprecise type message to an event parameter containing a message. This type information is insufficient and unsafe. As an example, imagine that an inattentive composer binds handler R'.handleReceive to an event triggered by a protocol module other than U'. In this case, handler R'.handleReceive may get a message with unknown or incorrect headers. This would result in a runtime error when R' tries to pop a header of the wrong type. Or even worse, if the type of the popped header happened to match the expected type (by chance), the execution would result in an erratic behavior.

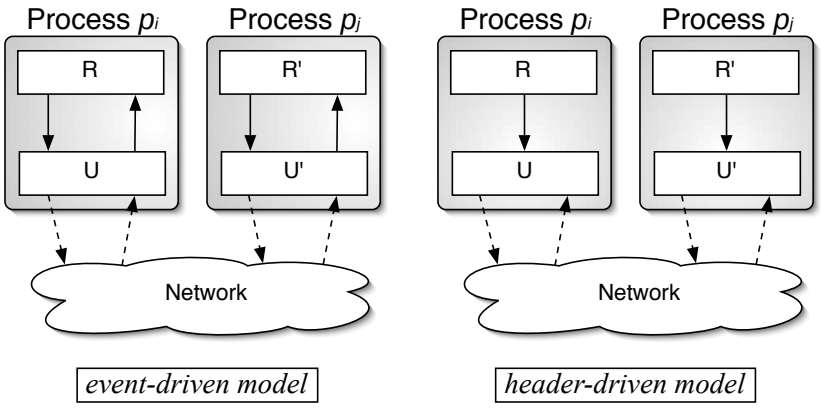


Figure 5.2: Typical bindings for peer interactions in the event-driven and header-driven models.

5.4 The Header-Driven Model

After presenting the event-driven model and discussing its inconvenience for protocol composition frameworks, we present our solution, which overcomes most of the problems discussed.

5.4.1 From Events to Headers: Overview of the New Model

Usually, protocol modules use one single entry point for incoming peer interactions, which is typically a handler called receive. For fairly complex protocols, different types of messages are expected to reach the receive handler. For this reason, the peer module inserts a tag into the header to distinguish the nature of the data in it. In those cases, the handler is structured as a big *switch* clause where the block of code executed depends on the tag value. In our model, we impose the declaration of one header per tag used. This way, we get rid of tags and, most importantly, a header name fully describes the structure of the data it contains. Additionally, big *switch*-based handlers are decoupled into several handlers.

As the message headers that a protocol module declares only make sense within its own code (and its peers' code), it looks natural to restrict the scope of that header to the only protocol module that uses it. An additional restriction in our model is that header names must be unique among all protocol modules in the composition. This is easy to achieve if, whenever a protocol module instance (say a) creates a header (say h), we have the framework automatically add the name of the module instance as a prefix of the header's name (i.e., a.h).

A composition satisfying these constraints has the following interesting property: if we inspect the sequence of header names in a message arriving from the network, we can approximately see the sequence of protocol modules (i.e. the route) that will handle the message on its way up. This means that there is no need for the composer to explicitly bind the upward flow of events. In other words, the message's header sequence drives its route up the protocol graph.

The event-driven model prevents us from exploiting this property. Therefore, instead of having events at the core of our interaction scheme, we should use *headers*. This is the essence of our proposal.

The essential ingredients of a *header-driven model* are headers and messages. A message is a list of headers. A header is a data structure and its name represents the data type. Protocol code is structured as a set of *header handlers*. A header handler declares a header name and the code that handles headers with that name. The declared header name is internal to the protocol module, unique throughout the whole (local) composition, and equal to the name of the symmetric header at other processes. Protocol execution is based on *message dispatch*. A message is dispatched in the following way: (1) its first header is popped from the message, (2) the unique header handler whose name matches the popped header is executed. The data contained in the popped header and the tail of the message are passed as arguments.

Compared to the event-driven model, we can say that (1) header handlers replace event handlers, (2) message dispatch replaces event triggering, and (3) the binding mechanism is dropped.

The composition of R and U in the header-driven model is depicted in the right part of Fig. 5.2. Solid arrows denote the dependencies needed between both protocols. Peer interactions are executed as follows.

Protocol module R starts a peer interaction by pushing a header into a message, whose name matches the header handler that R' is to execute when receiving the message, say R'.receive. This way, R is able to reflect in its code its a-priori knowledge of the handler used by R' to process the message. At this point, the message has to be (locally) sent to U. To do so, R simply pushes another header matching the sending handler at U, say U.send (this header contains the data U needs, such as destination process, etc.), and calls primitive dispatch(m), where m is the constructed message, resulting in m being routed to handler U.send. Then, first header of m (i.e., U.send) is popped and U.send receives the tail of m as a parameter. Header handler U.send pushes the header that U' will use to receive the message, say U'.receive, and transmits the resulting message m' through the network (e.g., using the framework's networking facilities). At this point, m' contains the sequence "U'.receive(...), R'.receive(...), ...". If m' is not lost, when it reaches the destination process it is simply dispatched by the framework. As the first header of m' is U'.receive, it will be automatically routed to its matching header handler. At the end of U'.receive, the message is again dispatched with primitive dispatch(m'). The first header is now R'.receive, so the message dispatch routes the tail of m' to the right handler of R': the one R pushed when creating the message.

The header-driven model is a better fit than the event-driven model for the ubiquitous peer interactions. The shortcomings of the event-driven model presented in the previous section are elegantly solved by the header-driven model:

- *Improved Up-going Mechanism.* Messages "know" where they are to be routed, especially when going upwards (see black arrows in Fig. 5.2). The event-driven model needs to bind the up-going receive handlers to the appropriate events, thereby becoming part of the protocol module's interface. In the header-driven composition, each up-going header handler is declared internally to its protocol module.

- *Event Routing Problem Disappears.* We have seen how the header-driven model routes up-going messages automatically to the right protocol instances. Therefore, if many protocols are using the same lower-level protocol, the routing problem presented above does not appear.

- *Correct Data Type in Headers Is Ensured.* When a header is created, it is given a name, which defines the type of the contained data. The header name chosen must have been declared as a unique header handler somewhere in the composition, otherwise the code is rejected at composition time (remember that uniqueness of header names is easy to achieve if they are prefixed by the name of the module instance that declares it). Hence, a message is a list of headers where each header has exactly one matching header handler in the local composition. Moreover, most distributed systems use identical protocol compositions at every process. If we make this assumption, a message transmitted over the network will reach a distant process that has the same header handler declarations. As a result, whether the message is dispatched

locally or transmitted to another process, each header will be handled by the one and only header handler whose name matches that of the header. It is impossible that a header has no matching handler (which would cause a run-time error), or is routed to a wrong handler (which would cause an erratic behavior).

As a conclusion, we can say that the header-driven model manages peer interactions better than the event-driven model. However, in the case of a composition where peer interactions are not frequent (which has never been our case), the event-driven model's simplicity remains a better fit.

5.4.2 Header-Driven Primitives

After an overview of the header-driven model, a detailed description of the model's primitives follows.

Message Construction. A header h($x_1, ..., x_n$) is a data structure represented by its parameters (x_i), which are statically typed, and its name (h). Header names are unique within a composition. A message is a list of headers, the head of the list is the *message destination* and the rest the *continuation*. Messages are built by pushing headers into the empty message [] using the right-associative :: operator. Two messages can be concatenated by operator @. These are some examples of messages:

```
m ← second(4+2, 5.0) :: first() :: []
m' ← third("hello") :: []
m" ← m @ m'
```

Message m is created by pushing header first, which has no parameters, to the empty message, then pushing header second, which contains two parameters, an integer and a floating-point number. Message m' is created by pushing header third with a string parameter to the empty message. Finally m", a concatenation of m and m', contains the sequence:

```
second(6, 5.0) :: first() :: third("hello") :: []
```

Messages, headers and header names are first-class values: they can be stored within a data structure. However, messages cannot be taken apart inside the handler code. The only way a message is disassembled is by message dispatching: the destination (i.e., the header at the top) is popped off the message and the matching header handler is invoked. We will get back to message dispatching later.

Header Handlers. For each of the three header names seen above, a matching header handler must be declared somewhere in the composition. A header handler is preceded by reserved word handler and has two parts: the header name it declares along with the formal parameters, and the protocol code to execute when the handler is invoked. Here is, for example, the declaration of handler second:

```
handler second(int someData1, float someData2) :: m is
  data3 ← someData1 + someData2
```

This handler declaration specifies that any header named second must contain two arguments of types integer and floating-point. Besides, it contains a line of code that adds the two arguments and stores the result within variable data3 (probably part of the protocol state).

Assume that the runtime decides to dispatch message m defined above to the handler matching its destination (second). Then, formal parameters someData1 and someData2 are assigned actual parameters 6 (i.e., $4 + 2$) and 5.0 respectively. Besides, formal parameter m is assigned the message continuation, i.e., first() :: []. Finally the body of the handler is executed. A header handler has the following properties:

- It is implicitly recursive: the header name it declares can be used in the body of the handler.
- As event handlers in the event-driven model, the invocation of a header handler is asynchronous in the sense that it does not return any value, it is only used for its side effect.

Message Dispatch. Message dispatch is the execution pattern at the core of our model. A message is dispatched using reserved word dispatch. Message dispatch is done as follows: (1) the message destination (the header at the top) is popped off the message, (2) the system looks up the unique header handler that corresponds to that header, and (3) it executes the header handler with the header's enclosed data and the message continuation (i.e., the remaining headers) as actual parameters. Protocols usually dispatch the message continuation at the end of the handler body. Dispatching the empty message [] has no effect (in some setups it could be configured to raise an exception).

This is an example of message dispatch:

```
state
  int i ← 0
handler repeat(int n) :: m is
  if n > 0 then
    dispatch (m @ (repeat(n-1) :: m))
handler count() :: m is
  i ← i + 1; dispatch(m)
handler start1() :: m is
  dispatch(repeat(100) :: count() :: [])
```

Assume we initially dispatch the message start1() :: []. Then, handler count executes 100 times, and the side effect is that state variable i equals 100 at the end. We can notice that handler count dispatches the continuation at the end: this is very common in the header-driven model, and makes routing of up-going messages easier. Now let us slightly change the example:

```
handler eat() :: m is
  /* Void handler body */
handler start2() :: m is
  dispatch(repeat(100) :: eat() :: count() :: [])
```

If we initially dispatch message start2() :: [], the inclusion of header eat() in the message breaks the loop, since this header handler does not dispatch the message continuation.

Remote Message Dispatch. We have just seen how message dispatch works locally. For inter-process communication (i.e., through the network) the framework offers a library that transmits messages from one process to another. We use reserved word rdispatch(p, m) for this purpose, where p is the identifier of the destination process and m is the message to be transmitted. If the message is not lost in the network, the effect of this primitive is to execute dispatch(m) within process p. The delivery guarantees provided by remote dispatch vary: it can be reliable, or it can drop messages. It depends on the properties we expect from the underlying network. In the remaining of this chapter, we assume that remote dispatch can lose messages.

An important feature of remote dispatching is the following: the fail-safe operation mode of (local) dispatching is kept if the compositions at all processes are symmetric. The reasoning is as follows. Assume an arriving message has a header h that was declared at protocol module instance a at the sending process. As the composition is symmetric, there is exactly one matching header handler defined within a's peer.

5.4.3 The Composition Model

So far, we have presented dispatching mechanisms, but nothing has been said about a suitable composition model. The composition model presented in this section is the second half of our header-driven proposal. This model is well adapted to the message dispatching interaction scheme presented above. It allows the use of parametric and hierarchical protocol modules. The three basic elements that constitute this model are modules (implementations), module abstractions (parametric implementations) and module interfaces.

Modules. Protocol module definitions are complete protocol modules ready to be composed with other modules. They are preceded by reserved word protocol. A protocol module A can contain (1) a state declaration, which is private to A, (2) a set of header handler declarations, and (3) nested protocol modules or module interfaces (see below) necessary to implement A. Here is an example protocol module:

```
protocol A is
  state
```

```
    int i ← 0
  handler h1() :: m is
    i ← i + 1
    dispatch(nestedProt.h2("tick") :: m)
  protocol nestedProt is
    handler h2(string message) :: m is
      print(message)
      dispatch(m)
```

This example defines protocol A. Its state is an integer variable, which is incremented every time a header with name h1 is dispatched by the matching handler. The protocol also contains nested protocol nestedProt, which declares header handler h2 (h2 prints the message contained in the header's argument). We can instantiate A using the new reserved word:

```
a ← new A
```

In this code a is an instance of protocol module A. To name a's headers h1 and h2 from anywhere in the composition, we have to write a.h1, resp. a.nestedProt.h2.

Module Interfaces. Module interfaces are similar to protocol module declarations, but they include no implementation. They are preceded by reserved word interface. They can be use to refer to services, but without forcing any particular implementation. They contain a set of header handler declarations (without body). Here is an example of interface:

```
interface MyInter is
  handler h1()
```

A protocol module that implements all the handlers declared by an interface, is said to implement that interface. This means that the protocol can be *plugged* anywhere the interface is used. For instance, protocol A above implements interface MyInter.

Module Abstractions. In the definition of protocol A above, a protocol is nested inside a more general protocol. Nested protocols usually implement lower level services that the enclosing protocol requires. In some cases, rather than committing to a particular implementation of the nested protocol, a nested interface can be used. This is the role of protocol module abstractions: they contain the implementation of a protocol, but some parts of this protocol refer to an interface, which is not implemented. Module abstraction definitions are preceded by the reserved words abstract protocol:

```
abstract protocol B(MyInter i) is
  state
    float f ← 1.0
  handler divide(float param1) :: m is
    f ← f / param1
```

dispatch(i.h1() :: m)

Abstract protocol B needs a protocol implementing interface MyInter in order to work: there are references to elements of the interface somewhere in B's code. Therefore, B cannot simply be instantiated as though it were a normal protocol, since part of its implementation is missing. At instantiation time, we need to provide B with a protocol implementing the needed interfaces, in our example: **new** B(A), since A implements MyInter. Furthermore, abstract protocols can also *implement* interfaces (although partially). Thus, we could need a chain of abstract protocols to instantiate an original one. As an illustration, imagine that protocol module nestedProt (within A) was actually an interface. In that case, A would be an abstract protocol. In order to instantiate B, we would need to plug A into B, and a protocol implementing nestedProt, say X, into A: **new** B(A(X)).

5.5 Header-Driven *vs.* Event-Driven

In this section, we present a toy example (a reliable message protocol using an unreliable message protocol) to demonstrate the power of the header-driven model with respect to the event-driven model. Figure 5.3 depicts the example.

In order to be fair, we assume that the event-driven model also features the composition model proposed in Section 5.4.3. In order not to clutter up the example, we have removed the logic of both protocols, i.e., the code responsible for keeping track of sent and urgent messages, retransmitting them if they are lost, eliminating duplicates, etc. All this code is not crucial to understand the compositional differences between the two models. The gaps (where parts of the code have been removed) is represented by dots (...). The basic data types used, are msg (message), bool (boolean) and pid (process ID). Reserved word this denotes the ID of the local process. We introduce primitive rtrigger in the event-driven example. Its behavior is analogous to rdispatch: rtrigger(p, e(args)) has event e(args) triggered at process p. Although this primitive is normally not present as such in event-driven frameworks, its use in this example makes the comparison easier.

Composition. In this example, both versions use the composition model presented in this chapter, so the way to compose them is very similar. Abstract protocol module R is parametric on interface Urel interface. Before instantiating R, the composer has to find another protocol that implements the Urel service, for instance U. So, to instantiate R, the composer plugs an instance of U into R:

u ← **new** U
r ← **new** R(u)

Figure 5.2 (page 64) illustrates this composition. It is a stack-based composition: only one instance of R is using u. The behavior of such a stack-based composition is the same in both models. However, the behavior will significantly differ if the composer decides to include another instance of R that uses u:

```
1: interface Unrel is
2:     event udeliver(msg m)
3:     handler hUSnd(pid dst, msg m)

4: protocol U is
5:     event udeliver(msg m)
6:     event fromNet(pid source, msg m)
7:     handler hURcv(pid source, msg m) is
8:         ...
9:         trigger(udeliver(m))
10:    handler hUSnd(pid dst, msg m) is
11:        ...
12:        rtrigger(dst, fromNet(this, m))
13:    bind(hURcv, fromNet)

14: interface Rel is
15:    event rdeliver(msg m)
16:    handler hRSnd(pid dst, bool urg, msg m)

17: abstract protocol R(Unrel u) is
18:    event rdeliver(msg m)
19:    event usend(pid dst, msg m)
20:    handler hRRcv(pid source, msg m) is
21:        (urg, m') ← pop(m)
22:        ...
23:        trigger(rdeliver(m'))
24:    handler hRSnd(pid dst, bool urg, msg m) is
25:        ...
26:        m' ← push(urg, m)
27:        trigger(usend(dst, m'))
28:    bind(u.hUSnd, usend)
29:    bind(hRRcv, u.udeliver)
```

event-driven model

```
1: interface Unrel is
2:     handler hUSnd(pid dst) :: m

3: protocol U is
4:     handler hURcv(pid source) :: m is
5:         ...
6:         dispatch(m)
7:     handler hUSnd(pid dst) :: m is
8:         ...
9:         rdispatch(dst, hURcv(this) :: m)

10: interface Rel is
11:    handler hRSnd(pid dst, bool urg) :: m

12: abstract protocol R(Unrel u) is
13:    handler hRRcv(pid source, bool urg) :: m is
14:        ...
15:        dispatch(m)
16:    handler hRSnd(pid dst, bool urg) :: m is
17:        ...
18:        dispatch(u.hUSnd(dst) :: hRRcv(this, urg) :: m)
```

header-driven model

Figure 5.3: Example protocol composition represented in both the event-driven and header-driven models.

```
r2 ← new R(u)
```

In the event-driven model, we run into an instance of the event-routing problem presented in Section 5.3.2. Even though r and r2 use different handlers (r.hRRcv and r2.hRRcv), they both bind their handler hRRcv to the same event u.udeliver. If a peer module of r starts a peer interaction with r, r2 will also receive the message, which is not the intended behavior. In the header-driven model, everything still works fine. Protocol instances r and r2 use different up-going headers: r.hRRcv and r2.hRRcv. Besides, every up-going message contains the good header, since it was pushed at the peer protocol instance. As a result, u will dispatch all messages correctly at the end of handler hURcv.

Implementation of Unreliable Protocol U. At the *unreliable* level, an interface (Unrel) and a protocol module (U) are defined. At first sight, the interface appears simpler in the header-driven version. The reason is the up-going event udeliver, which is not needed in the header-driven version. In the event-driven version, the upwards path starts with the triggering of a fromNet event. This occurs when a remote process has executed rtrigger on the fromNet event. The complete path is: fromNet triggering, bound to handler hURcv, which triggers event udeliver, the latter being part of the interface. The higher level protocols using an instance of U need to bind a handler to this event. In the header-driven model, the interaction path is simpler. First, event declarations and bindings are not necessary. The upward path starts with a dispatch of a message as a result of a remote rdispatch of that message. The message already contains in the top header the handler to call first at the receiving process: hURcv(source). So the message is simply dispatched. When the handler is done with the message, it does not need to find out where to forward it up the graph: the message itself knows where it has to go (again, the handler matching its top header). So, again, the message is simply dispatched.

Implementation of Reliable Protocol R. At the *reliable* level, again an interface (Rel) as well as a protocol module (R) are defined. As before, the interface is simpler in the header-driven style, where up-going event rdeliver is dropped. The handler declared in the interface for sending messages, hRSnd, contains three parameters. The first and the third ones do not need explanation. The second, urg, is a boolean that indicates the urgency of the message, i.e., whether it should take priority over other in-transit messages. Its value is used in the protocol logic (which is omitted). Besides, urg has to be transmitted to the peer reliable protocol, which is why it is interesting in our example. In order to transmit the urg value to the peer protocol module, the event-driven version needs to push it (hence function push in the code) into the message as a header, losing the type information. Thus, when the message arrives at the peer protocol module, its headers contain no type description. Handler hRRcv *guesses* that the first header is the urgency value and pops it from the message. If the message happens to have a header of a different type (e.g., because of a wrong binding), it can result in a runtime type-mismatch error. In the header-driven model, we can modify the signature of handler hRRcv at our will, since it is not exposed in the interface (this is not possible in the event-driven version because the signature of the handler must match that of the event it is bound to). We modify hRRcv adding the urgency variable as a parameter of the handler. Now, the compiler can help us checking the type of this argument. To send the message to the peer protocol module, handler hRSnd pushes two headers into the message. The first header will be dispatched by the handler that locally processes the message (u.hUSnd). The second is the handler that will process the message at the peer module (r.hRRcv). Finally, as in U, up-going messages are simply dispatched: the routing information is within the messages themselves.

Code Readability. Even for such a simple composition, a glance at the code is enough to notice the improved readability of the header-driven model in comparison to the event-driven model. The reasons are several:

- *No Bindings*. Binding events to handlers introduces a level of indirection, which complicates the code. In the header-driven model, the bindings are implicit: a header is bound to the unique handler whose name matches.

- *Starting Peer Interactions Is Clearer*. To start a peer interaction, the event-driven model simply triggers an event (see left part of Fig. 5.3, line 27), where there is no trace of which handler will be used at the peer protocol module to process the message. Besides, we have to look up in the event bindings (line 28) to find out which local lower-lower protocol will process the message. In the header-driven model, only by looking at the message dispatch that starts the peer interaction (see right part of Fig. 5.3, line 18), we can find (1) the local lower-level handler that will process the message, and (2) the handler at the peer module that will receive the data. So, line 18 can be interpreted as r using local u.hUSnd handler to call hRRcv at r's peer.

- *No Explicit Path for Up-Going Messages*, which simplifies interfaces. Besides, the code responsible for sending upwards an up-going message is also simplified. As an example, compare lines 21 and 23 of the event-driven model with line 15 of the header-driven model.

- *No Need for Message Tags*. If there is a single entry point (handler) for different types of up-going messages, we need to define tags in the event-driven model. This is a typical solution:

  ```
  handler receive(pid source, msg m) is
    (tag, m') ← pop(m)
    if (tag = FOO) then (float f, m") ← pop(m'); ...
    else if (tag = BAR) then (int h, m") ← pop(m'); ...
    ...
  ```

  While this works, there is a big handler structured as a set of code blocks. Each code block processes messages with a given tag. In the header-driven model, as the up-going route is not included in the interface, we can split the big generic handler into several specialized ones, one for each different tag:

  ```
  handler foo(pid source, float f) :: m is ...
  handler bar(pid source, int h) :: m is ...
  ```

5.6 Conclusion

In this chapter, we have seen the main features of an event-driven framework, and discussed its drawbacks. We have presented a new way to look at protocol module interactions: the header-driven model. It has several advantages: better routing

scheme, more rigorous type information for messages, better protocol readability, and less compositional problems.

In the header-driven model, the interaction core is shifted from event triggering to message dispatching. Therefore, protocols programmed in the event-driven model are incompatible with those programmed in the header-driven model.

Our model was implemented as a syntactic extension to the OCAML programming language [Ler00] under the name NUNTIUS [Gen04]. Experiments with this prototype showed that our programming model associated to OCAML's module system fulfills its compositional promises.

Part II

Advances in Modular Group Communication

Chapter 6

System Models, Protocol Specifications and Toolkits

This chapter and the rest of Part II are devoted to group communication protocols and the contributions that this thesis makes to this field. There has been a large body of research on group communication in the last two decades. Various system models have been proposed in an attempt to represent real systems in a simplified manner. For instance, some models consider the possibility of adding fresh processes to the system at any moment, while others allow malfunctioning processes to recover. We have conducted our research in several of these system models, whose results are presented in the different chapters that follow.

The present introductory chapter offers a description of the research background: we first describe the system models defined in the group communication literature, we then present specifications of well-established group communication protocols, and finally we survey the group communication toolkits appeared in the last years that are most relevant to our research.

6.1 Introduction

Chapters 7 and 8 put forward our contributions to group communication research. In Chapter 1, we described the evolution of group communication toolkits: from monolithic to modular, and how general-purpose protocol composition frameworks were progressively developed to structure modular design of group communication protocols. This part of the thesis is devoted to the algorithmic implications of modularity in group communication design. As group most communication protocols offer rather elaborate properties to the application, their modular design normally includes a number of building blocks. This chapter presents the specification of those building blocks that are necessary to understand our contributions.

Before undertaking the design of a group communication architecture, one needs to make several basic assumptions that reflect the way the underlying system works: its guarantees and weaknesses. This is known as the *system model*. The

choices made at this point will impact the design enormously, to the extreme that some group communication problems may become unsolvable under certain system models. On the other hand, assuming unrealistically strong guarantees from the system model may lead to trivial algorithms that are barely useful in reality.

Some guarantees usually defined in a system model are for instance: the asynchrony of the system (how long it takes to transmit a message on the wire), the nature of failures (crashes, malicious behavior), etc. Section 6.2 goes over the most significant aspects of a distributed system model.

As previously said, a modular group communication protocol is composed of a number of building blocks. Some of them are basic protocols for low-level message exchange, others are more complex. In Section 6.3 we present the specifications of these protocols, from most basic (bottom-most blocks, near the network) to most complex group communication-related building blocks (at the top of the composition).

Finally, Section 6.4 is a survey of traditional group communication toolkits from the point of view of their (modular or monolithic) architecture.

6.2 System and Failure Models

The system (and failure) models are a key issue when specifying or implementing a group communication algorithm. The impact of the system model chosen is crucial: an algorithm may work properly in a system model, yet it can be useless in another. Moreover, the system model chosen can determine the maximum degree of fault tolerance that can be achieved. There are several aspects that define a system model. Usually, defining a system model entails making choices for all the aspects described in this section.

In all system models considered in the thesis, we consider the system to be composed of a finite set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$. There is no shared memory between any pair of processes: each process has its own memory address space. Every pair of processes is connected by a bidirectional network channel. Communication among processes is performed exclusively via message exchange through a low-level network service. The properties of this network service depend on the particular assumptions made for the different system models.

6.2.1 Synchrony

The synchrony of a system describes how good a system is to deliver messages in a timely manner. It also tells whether there is a limit on the relative speed of the processes that constitute the system.

Synchronous Model. In a synchronous system model, there is an upper bound on the transmission time of all messages. This upper bound is known a priori and its knowledge can be exploited. In [Lyn96], the synchronous system is described

as a set of synchronous rounds where (1) each process sends a message (which can be *null*) at the beginning of a round, (2) messages are received and the end of the same round, and (3) upon reception of these messages, each process takes a computation step that leads to the next round. For such synchronous rounds to be feasible, among other things, their duration should be longer than the bound on message transmission time.

Asynchronous Model. In an asynchronous system model, message transmission time has no upper bound, that is, messages can take arbitrarily long to reach the destination process. Additionally, the relative speed of processes is not bound either: a process can be arbitrarily slow to process a message with respect to another process. As a result, if a process p sends a message to another process q, it can take arbitrarily long to q to receive the message, even if the message is not lost and neither p or q crash.

Other Synchrony Models. In many cases, the synchronous model is too strong an assumption because it does not allow any message to be late. On the other hand, a fully asynchronous model is too weak for some algorithms to be solvable, because it allows situations in theory where all messages are very late. Real systems are typically between these two extremes: messages are usually timely, and processes have similar speeds but there are some unstable periods when messages are late an processes are overloaded (and thus slow).

This has led some researchers to define new synchrony models that try to grasp the eventual asynchrony of a system that is supposed to behave synchronously in the general case. The *partial synchronous model* [DLS88] considers that the message transmission time and relative speed of processes are both bound, but this bounds are either not known or they only hold after some unknown time. The *timed asynchronous model* [CF99], makes assumptions on maximum message transmission time and considers late messages as *omission failures*, which are explained in Section 6.2.2.

Focus of this Thesis. In this thesis we always assume the system to be asynchronous. The main reason is that protocols that work under this assumption are more general and time-free. They make progress when the real network offers some degree of synchrony, and block (without doing *bad things*) during periods when the network is fully asynchronous. The main drawback is that this asynchronous model is so weak that many problems are unsolvable [FLP85, CHTCB96, CT94, Ric96]. For this reason, we augment the asynchronous system model with unreliable failure detectors [CT96], which are described in Section 6.3.2.

6.2.2 Failures

Processes behave in a distributed system accordingly to their algorithms most of the time. However, dependable distributed systems have to deal with *failures*. A

failure occurs when a process deviates from its normal behavior in some way. The possible manners in which a process can misbehave are the following:

- *Crash Failure.* A process suddenly stops its operation. This is the most benign failure type.
- *Send Omission Failure.* A process is supposed to send a message, according to its algorithm, but it does not send it.
- *Arbitrary or Byzantine Failure.* Byzantine failures [LSP82], the most general type, assumes that processes may fail by behaving arbitrarily, even maliciously.

Fault-tolerant systems make an assumption on the number of processes that can fail. This assumption is represented by constant f, which denotes the maximum number of processes that can misbehave in a given run. The value of f is important when specifying algorithms because a slight variation of this constant can imply the impossibility to solve a given problem.

Focus of this Thesis. This thesis only considers crash failures, which are the ones most frequently found in the literature, although we believe some of our results could be (indirectly) extended to more general failures. Nevertheless, Chapter 10 considers message corruption and send omission failures in practice, but only in order to assess the system's ability to transform them into clean crash failures.

6.2.3 Groups

Group communication algorithms are based on the notion of *group of processes*. A group of processes is a finite set of processes that may or may not change during system lifetime, and should coincide with set Π defined above.

Static Groups. In a model with static groups, or *static model*, the initial group of processes Π is fixed at system start-up time and remains unchanged throughout the whole run. This is the simplest case, which does not require a group membership protocol. The main drawback is the impossibility of adding new processes to the group, for instance, to replace the ones that have crashed.

Dynamic Groups. The model with dynamic groups, or *dynamic model*, allows processes to be added to or removed from the group after system start-up time. Adding or removing processes must be done in an orderly manner. For that purpose, a *group membership service* is defined (its specification is presented in Section 6.3). Group communication toolkits can be further classified depending on the way they deal with network partitions. A *network partition* is a disconnection of a subset of processes form the rest: the network channels between processes within

the same partition work properly, whereas channels that link processes in different partitions are (temporarily) broken. There are mainly two trends in state-of-the-art group communication toolkits:

- *Primary Partition.* In the presence of a network partition, only processes located in one partition, called *primary partition*, are allowed to continue operation. The primary partition contains the majority of processes.[1] Processes located in other partitions should block as long as they can not communicate with the primary partition (i.e., as long as the network partition is not fixed).
- *Partitionable.* If a network partition occurs, all processes are allowed to continue operation along with other processes in the same partition. The price to pay is that the state of processes at different partitions can diverge, possibly causing inconsistencies at the application level. Thus, the application needs to perform an ad-hoc *merge* operation when the partition is healed.

Focus of this Thesis. This thesis makes contributions in both the static and dynamic systems. The new architecture presented in Chapter 7 uses primary-partition dynamic groups. The system model assumed in Chapter 8 uses static groups.

6.2.4 Recovery Capabilities

The point in time at which a process fails is unpredictable by nature. At the beginning, all processes are supposed to behave correctly, even the faulty ones.[2] Consider a system where processes can only fail by crashing. When a process crashes one may consider the possibility that it *recovers* (i.e., it is restarted) after some time and rolls back to a state checkpointed to disk before the crash. Depending on whether processes have this recovery capability, we can classify a system with crash faults into *crash-stop* (also called crash-no recovery) and *crash-recovery*. As some basic definitions are different depending on whether the system is crash-stop or crash-recovery, we present them separately.

6.2.4.1 The Crash-Stop System Model

Process Crash. A process that crashes stops its operation permanently and never recovers. A process is faulty in a run if it crashes in that run. The state of a process is kept in main memory only. If a process crashes, its state is lost and can not be recovered.

Correct and Faulty. For a given run (i.e., execution of the system), if a process does not fail, we say it is *correct*. On the other hand, if a process does fail, we say it is *faulty*.

[1] The definition of this *majority* depends on the particular system, but guarantees that two majorities always intersect.

[2] A faulty process is considered faulty even when its behavior is still normal

6.2.4.2 The Crash-Recovery System Model

Crash and Recovery. Processes can crash and may subsequently recover. We consider system start-up time as an implicit *recover* event. In any process' history, a *recover* event happens always immediately after a *crash* event, except at system start-up time. Moreover, the only event that can happen immediately after a *crash* event (if any) is a *recover* event.

Up and Down. A process q is *up* within the segments of its history between a *recover* event and the following *crash* event. If no *crash* event occurs after the last *recover* event in q's history, then q is up forever from its last *recover* event on. In this case, we say q is *eventually always up*. A process q is *down* within the segments of its history after a *crash* event until the next *recover* event (if such an event exists).

Good and Bad Processes. A process is *good* if it is *eventually always up*. A process is *bad* if it is not *good*. In other words, a process is *bad* if it (a) eventually crashes and never recovers or (b) crashes and recovers infinitely often.

6.2.4.3 Focus of this Thesis

Most of group communication literature does not consider the possibility of process recoveries. We pursue this path in Chapter 7, where processes do not recover. However, a crashed process can be re-admitted to the group under a fresh new identity.[3] Therefore, the re-admitted process loses all its state and a state transfer is needed. On the other hand, Chapter 8 contributes with a sound specification of atomic broadcast (see Sect. 6.3.5) in the crash-recovery model.

6.3 Protocol Specifications. From Fair-Lossy Channels to Group Communication

The remaining chapters of Part II deal with a number of group communication problems. In this section we present their specification. Most modular group communication protocols need some lower-level protocols in order to fulfill their specification. We present the specifications for all of them here: we proceed from the most basic problems to group communication.

6.3.1 Communication Channels

In our system model, every pair of processes is connected by a bidirectional network channel. Every message broadcast by the application in a run is unique (it is usually attached the pair $\langle$ *process ID*, *sequence number* $\rangle$) and taken from set $\mathcal{M}$ of

[3] This is usually know as a new *incarnation* of a process.

all possible messages (universe of messages). A communication channel provides two communication primitives: *send*(m, p) and *receive*(m, p), where $m \in \mathcal{M}$ and $p \in \Pi$. We say that process p sends message m to destination process q if p executes *send*(m, q). Likewise, we say that p receives message m from process q if p executes *receive*(m, q).

We define the following properties regarding communication channels.

Property 6.3.1 NO-CREATION. *If process q receives message m from p, then p has sent m to q.*

Property 6.3.2 NO-DUPLICATION. *If process q receives message m from p, q receives m from p at most once.*

Property 6.3.3 NO-LOSS. *If process p sends message m to q, and both processes are correct, then q eventually receives m from p.*

Property 6.3.4 FAIR-LOSS. *If process p sends an infinite number of messages to q and q is correct, then q receives an infinite number of messages from p.*

There are different levels of channel reliability in the distributed systems literature. Here, we define *quasi-reliable channels* and *fair-lossy channels*. A communication channel is *quasi-reliable* if it fulfills Properties 6.3.1, 6.3.2, and 6.3.3. Some authors define *reliable channels*, which are stronger than quasi-reliable channels, by not requiring process p to be correct in Property 6.3.3, but it is not realistic from a practical point of view. A communication channel is *fair-lossy* if it fulfills Properties 6.3.1, and 6.3.4.

Stubborn channels were introduced in [GOS98] as a helpful abstraction for the crash-recovery model, where Property 6.3.3 does not make sense. A communication channel is *stubborn* if it fulfills Property 6.3.1 and the following one:

Property 6.3.5 STUBBORNNESS. *Let p and q be two good processes. If p sends a message m to q and indefinitely delays sending any further message to q, then q eventually receives m.*

The *indefinite delay* requirement expresses that, as soon as p sends another message m' to q (after sending m), there is no more obligation for q to receive m. However q *is* required to receive m as long as no p does not send any further message to q after m.

6.3.2 Unreliable Failure Detectors

The notion of unreliable failure detectors was first formalized by Chandra and Toueg [CT96] in the context of process crashes. A failure detector can be seen as a set of modules, with one module FD_i attached to every process p_i. A process p_i can query its failure detector module FD_i about the state of other processes; the

output can be *correct* or *faulty*. The information returned by a failure detector can be incorrect: a failure detector module may yield the output *correct* for a process that has crashed and *vice versa*. These inaccuracies in the output lead us to say that a failure detector *suspects* a process to have crashed. Failure detectors may also give inconsistent information: for instance FD_i can suspect a process p at a given time t, while FD_j does not suspect p at time t.

Failure detectors are defined in terms of a completeness and an accuracy property. The completeness property describes the ability of the failure detector to detect crashed processes. The accuracy property restricts the way a failure detector can incorrectly suspect processes that have not crashed. We consider the following properties [CT96]:

Property 6.3.6 STRONG COMPLETENESS. *Every incorrect process is eventually suspected by every correct process.*

Property 6.3.7 STRONG ACCURACY. *No correct process is ever suspected by any correct process.*

Property 6.3.8 EVENTUAL WEAK ACCURACY. *There is a time after which some correct process is never suspected by any correct process.*

We now define two failure detectors: the $\mathcal{P}$ failure detector (perfect) satisfies Strong Completeness and Strong Accuracy, and the $\diamond\mathcal{S}$ failure detector (eventually strong) satisfies Strong Completeness and Eventual Weak Accuracy. The asynchronous model augmented with one of these failure detectors is strong enough to solve all the group communication problems considered in this thesis. However, $\mathcal{P}$ is too strong an assumption for many real systems.

6.3.3 Uniform and Non-Uniform Protocols

Group communication problems are specified by a set of properties they have to fulfill. Each of these properties restrict the behavior of one or several processes. A non-uniform property only mentions correct processes, in other words, such a property does not restrict the way faulty processes behave. In contrast, a uniform property states that both correct and faulty processes (before they crash) must respect the required behavior. Therefore, uniform properties are strictly stronger than their respective non-uniform versions. Non-uniform protocol specifications contain exclusively non-uniform properties. Uniform protocol specifications contain mostly uniform properties, but may also contain some non-uniform properties if the uniform version does not make sense (e.g., because it is impossible to guarantee).

This distinction is interesting from a practical point of view [SS93]: it is usually more efficient to implement non-uniform properties, allowing faulty processes to behave erratically. Hence, from a performance viewpoint, it is usually a better

choice to implement a non-uniform specification, as long as the application on top tolerates it. Otherwise, a slower uniform version of the protocol is required.

In the remaining of this chapter, we only give uniform specification of problems. Obtaining the non-uniform version is easy: replace every reference to *process* by *correct process*. For example, the non-uniform version of Property 6.3.11 (see below) becomes the following:

Property 6.3.9 AGREEMENT. *Two* correct *processes never decide differently.*

This non-uniform version allows two processes to decide differently as long as one of then is faulty (i.e., will crash in the future).

The distinction between uniform and non-uniform protocols is well understood in the crash-stop model. In contrast, this is not the case in the crash-recovery model. Chapter 8 is in part an attempt to clarify the concept of uniformity in the crash-recovery model.

6.3.4 Consensus

Consensus [Fis83] is an important abstraction in distributed computing. Moreover it can be used as a building block to solve atomic broadcast [CT96] (see Sect. 6.3.5) as well as other problems. We define primitives *propose*($value$) and *decide*($decision$). We say that a process p *proposes* initial value v if p executes *propose*(v). Likewise, we say that a process p *decides* value v if p executes *decide*(v). Informally, consensus ensures that processes reach an agreement on a value proposed by one of them. More formally, uniform consensus is defined in the crash-stop model by the following properties:

Property 6.3.10 UNIFORM INTEGRITY. *A process decides at most once.*

Property 6.3.11 UNIFORM AGREEMENT. *Two processes never decide differently.*

Property 6.3.12 UNIFORM VALIDITY. *If a process decides, the decision value is the initial value of some process.*

Property 6.3.13 TERMINATION. *All correct processes eventually decide.*

Uniform consensus is also defined in the crash-recovery model [ACT00]. The properties are analogous to the ones above.

6.3.5 Broadcast Protocols in the Static Model

Specifications of broadcast services in the static crash-stop model [HT94] are widely accepted. The specifications in this section are defined for this model. In the dynamic and crash-recovery models, specifications of broadcast protocols have not reached (yet) an acceptable level of maturity. Chapter 7 presents an architecture

based on a novel specification of broadcast services for dynamic groups. This novel specification is an attempt to clarify protocol specifications in the dynamic model and was introduced by Schiper in [Sch06]. On the other hand, no satisfactory specification for atomic broadcast (the most relevant form of broadcast) has been proposed to this day in the crash-recovery model. Chapter 8 proposes a specification (and an implementation) for atomic broadcast in that model.

Reliable Broadcast. We define primitives *rbcast*(msg) and *rdeliver*(msg), where $msg \in \mathcal{M}$. We say that a process p reliably broadcasts (or simply *rbcasts*) message m if p executes *rbcast*(m). Likewise, we say that a process p reliably delivers (or simply *rdelivers*) message m if p executes *rdeliver*(m). Informally, reliable broadcast is often used to implement other broadcast protocols with stronger properties. It ensures that messages are rdelivered either by all correct processes or by none, even if the sender crashes while rbcasting the message. However, it does not enforce any order in rdelivered messages. More formally, reliable broadcast is defined by the following properties [HT94]:

Property 6.3.14 VALIDITY. *If a correct process p rbcasts message m, then some correct process eventually rdelivers m.*

Property 6.3.15 UNIFORM INTEGRITY. *Every process rdelivers a message m at most once and only if m was previously rbcast by some process.*

Property 6.3.16 UNIFORM AGREEMENT. *If a process rdelivers a message m then every correct process also rdelivers m.*

Atomic Broadcast. We define primitives *abcast*(msg) and *adeliver*(msg), where $msg \in \mathcal{M}$. We say that a process p atomically broadcasts (or simply *abcasts*) message m if p executes *abcast*(m). Likewise, we say that a process p atomically delivers (or simply *adelivers*) message m if p executes *adeliver*(m). Informally, atomic broadcast (also called total order broadcast) is a stronger form of reliable broadcast where all messages are adelivered in the same order at every process. More formally, atomic broadcast is defined by the three properties that define reliable broadcast[4] and the following additional property [HT94]:

Property 6.3.17 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.*

Generic Broadcast. We define primitives *gbcast*(msg) and *gdeliver*(msg), where $msg \in \mathcal{M}$. We say that a process p generic broadcasts (or simply *gbcasts*) message m if p executes *gbcast*(m). Likewise, we say that a process p generic delivers (or

[4] Obviously, *rbcast* is renamed to *abcast* and *rdeliver* to *adeliver* in these properties.

simply *gdelivers*) message m if p executes *gdeliver*(m). We define conflict relation $\mathcal{C} : \mathcal{M} \times \mathcal{M} \rightarrow$ boolean. $\mathcal{C}$ is non-reflexive and symmetric. This conflict relation is based on semantics of messages and provided by the application at system start-up time. Its meaning is the following: if two messages conflict, they have to be delivered in the same order at all processes (just as atomic broadcast would); if they do not conflict, they can be delivered in any order. Informally, generic broadcast [PS99, PS02, ADGFT00] is a more flexible form of broadcasting messages. For example, if conflict relation $\mathcal{C}$ is void, then generic broadcast coincides with reliable broadcast. On the other extreme, if $\mathcal{C}$ is such that any two messages conflict, then generic broadcast is equivalent to atomic broadcast. More formally, generic broadcast is defined by the three properties that define reliable broadcast (with the obvious primitive renaming) and the following additional property:

Property 6.3.18 UNIFORM GENERALIZED ORDER. *For any two processes p and q and any two messages m and m', if m conflicts with m' and p gdelivers m before m', then q gdelivers m' only after having gdelivered m.*

6.3.6 Broadcast Protocols in the Dynamic Model

In a dynamic group model, a protocol maintaining a set of currently active processes is needed. This is the task of group membership. This set can change with new members joining and old members leaving. Each process has a *view* of this set, and when this set changes, the membership protocol is required to report the change at each process by installing a new view.

There are mainly two flavors of group membership in the literature: primary partition membership and partitionable membership (see Sect. 6.2.3). As the work carried out in this thesis focuses on the primary partition version, we present it here, and refer the reader, e.g., to [CKV01] for a formal definition of the partitionable membership.

Traditionally, specifications of primary-partition group membership have been defined so that the sequence of views installed reflects the processes that are up at each moment. This has led to fairly complicated specifications of group membership. In [ST06], the authors consider the group membership problem as the combination of two orthogonal subproblems: (1) determining the set of processes that are currently up, and (2) ensuring that processes agree on the successive values of this set. For solving problem (1), one does not really need group membership, but rather a failure detector that monitors processes and informs group membership when a process has been suspected for a sufficiently long time. For solving problem (2), we can proceed in two steps: first specify *set membership* and then a particular case of it: *group membership*.

Set Membership. The goal of set membership is that all processes agree on a global sequence of views, which contain a set of items that are not necessar-

ily processes.[5] A view V consists of a unique identifier $V.\mathrm{id}$ and a set of items $V.\mathrm{members}$ drawn from set $\mathcal{S}$ of arbitrary elements. For simplicity, we take view identifiers from the set of natural numbers. We define primitives *add*(x), *remove*(x), and *new_view*(V), where $x \in \mathcal{S}$ and V is a view. We say that a process p requests item x to be added to the set if p executes *add*(x). Likewise, we say that a process p requests item x to be removed to the set if p executes *remove*(x). Additionally, we say that p installs view V if p executes *new_view*(V). All actions at p after installing view V, up to and including the installation of another view V', are said to occur in V. The specification of set membership is satisfied if there exists a *global sequence* of views H, such that the following properties hold:

Property 6.3.19 VIEW SEQUENCE AGREEMENT. *For every process p, the sequence of views it installs during its whole execution is a subsequence of H.*

Property 6.3.20 INTEGRITY. *Assume a process p installs view V' in V. If an item x is in $V'.\mathrm{members}$ but not in $V.\mathrm{members}$, then some process requested x to be added to the set.*

Conversely, if an item x is in $V.\mathrm{members}$ but not in $V'.\mathrm{members}$, then some process requested x to be removed from the set.

Property 6.3.21 VIEW INSTALLATION. *Every view $V \in H$ is installed by every correct process.*

Property 6.3.22 OPERATION EXECUTION. *If a correct process requests item x to be added to the set in view V and $x \notin V.\mathrm{members}$, then there exists a view V', occurring after V in H, such that $x \in V'.\mathrm{members}$.*

If a correct process requests item x to be removed from the set in view V and $x \in V.\mathrm{members}$, then there exists a view V', occurring after V in H, such that $x \notin V'.\mathrm{members}$.

Group Membership. Once set membership is specified, we can define group membership as a special case of set membership where elements of $\mathcal{S}$ are processes. Besides, the specification of group membership can be specialized in the following manner:

- Usually, applications do not require processes to install views they do not belong to. Thus, we relax Property 6.3.21 by stating it as follows:

  Property 6.3.23 VIEW INSTALLATION. *A correct process must install every view it belongs to.*

- We can add the following property to prevent processes outside the group from issuing requests to add/remove[6] processes:

[5] They can be, e.g., the set of employees, or the addresses of a mailing list.

[6] In group membership, primitive *add* can also be called *join*.

Property 6.3.24 PERMISSION TO REQUEST A VIEW CHANGE. *A process p can request can issue a request to add/remove a process in a view V only if $p \in V$.*members.

- Sometimes, some correct process p is excluded from the view. In this case, group membership should no more have the obligation to execute p's pending requests to add/remove processes. For this purpose, we can restate Property 6.3.22 in this way:

  Property 6.3.25 OPERATION EXECUTION. *If a correct process p requests process q to be added to the group in view V and $q \notin V$.*members*, then there exists a view V', occurring after V in H, such that either (a) $q \in V'$.*members*, or (b) $p \notin V'$.*members.

  *If a correct process p requests process q to be removed from the group in view V and $q \in V$.*members*, then there exists a view V', occurring after V in H, such that either (a) $q \notin V'$.*members*, or (b) $p \notin V'$.*members.

View-Synchronous Broadcast. Group membership is only useful if one defines how the sending and delivery of broadcast messages interact with view changes. A variety of multicast protocols can be defined, with different ordering guarantees, just like in systems in the static group model. Here, we define view-synchronous broadcast [BJ87, CKV01].

We define primitives *VS-cast*(msg) and *VS-deliver*(msg), where $msg \in \mathcal{M}$. We say that a process p *VS-casts* message m if p executes *VS-cast*(m). Likewise, we say that a process p *VS-delivers* message m if p executes *VS-deliver*(m). Informally, view synchronous broadcast ensures that (1) the messages sent in a view V are delivered in the same view V, and that (2) all correct processes in V deliver the same set of messages in V. More formally, view synchronous broadcast is defined by the following properties (note the similarities to the properties that define reliable broadcast in Sect. 6.3.5):

Property 6.3.26 INITIAL VIEW. *Every VS-broadcast and VS-deliver occurs in some view.*

Property 6.3.27 UNIFORM INTEGRITY. *For any message m, every process VS-delivers m at most once, and only if m was VS-broadcast by some process.*

Property 6.3.28 VALIDITY. *If a correct process VS-broadcasts m then it eventually VS-delivers m.*

Property 6.3.29 SENDING VIEW DELIVERY. *If a process p VS-delivers message m in view V, then the sender of m VS-broadcasts m in view V.*

Property 6.3.30 VIEW SYNCHRONY. *If processes p and q install the same view V in the same previous view V', then any message VS-delivered by p in V' is also VS-delivered by q in V'.*

Property 6.3.31 AGREEMENT. *If correct processes p and q install view V as their last view, then any message VS-delivered by p in V is also VS-delivered by q in V*

In order to implement Property 6.3.29 without discarding messages from correct processes, processes must stop sending messages for some time before a new view is installed (the group membership protocol must tell the application when it should stop sending messages). One can avoid this blocking period by replacing Property 6.3.29 with a weaker property [FvR96, KM00]:

Property 6.3.32 SAME VIEW DELIVERY. *If processes p and q both VS-deliver message m, they VS-deliver m in the same view V.*

6.4 Group Communication Toolkits in the 90s

In [CKV01], Chockler *et al* describe a comprehensive set of specifications of group communication protocols, which correspond to the most popular implementations. These specifications can serve as a unifying framework for the classification, analysis and comparison of the group communication toolkits that have been implemented over the last 20 years.

In this section we present the architecture of existing group communication toolkits. Since it is not possible, and also not really worth, to present the architecture of all group communication systems that have been implemented, we have selected here the most representative architectures, and refer the reader to [CKV01] for a synthesis of the rest.

We have divided this section into two parts: monolithic and modular toolkits. As explained in Chapter 1, monolithic systems do not allow the system to be easily customized to the user needs; modular systems allow the user, using off-the-shelf components, to build the protocol stack that fits particular needs. Monolithic systems do not require any protocol composition framework, whereas modular toolkits use either an ad-hoc protocol composition framework or a general-purpose one as the ones presented in Section 2.3.

Among all existing monolithic toolkits, we have chosen to present Isis [BJ87, Bir93], Phoenix [Mal96], RMP [WMK94, Mon94], and Totem [AMMS+95]. On the other hand, among modular toolkits the most representative one is Ensemble [Hay98]. There are many other group communication toolkits but their architecture overlaps with the ones presented here. Transis [DM96], Relacs [BDGB95], and Newtop [EMS95] architectures overlap with Totem and Ensemble. The JavaGroups toolkit [Ban02] is strongly inspired by Ensemble (it can even be configured to use an Ensemble stack). The group communication protocol suite implemented

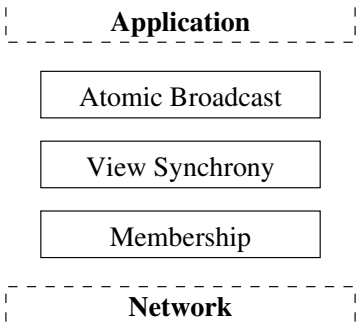


Figure 6.1: Isis architecture

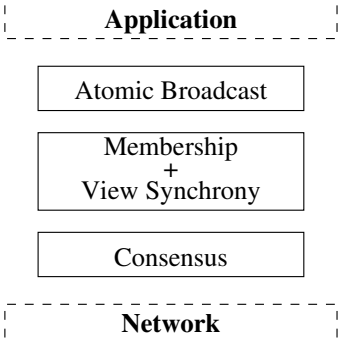


Figure 6.2: Phoenix architecture

in the Appia framework [MPR00] is also strongly inspired by Ensemble. The membership service presented in [HS98] uses a token based approach as in Totem or RMP.

6.4.1 Monolithic Toolkits

Isis. Isis was the first system to propose group communication [BJ87, BSS91]. It is a monolithic *primary partition* system, i.e., when a network partition occurs, the computation can only proceed in one partition of the network, called the *primary* partition. The Isis architecture is depicted in Figure 6.1. The main layers are the following:[7]

- The *group membership* layer, which is responsible for maintaining the membership of groups. This layer handles *joins* (request to join the group) and *leaves* (request to leave the group). The layer also excludes processes that are suspected to have crashed. As seen in Sect. 6.3.6, the group membership layer ensures that processes deliver the successive views in the *same total order*.

- Group membership does not provide any semantics for communication. For that reason, the group membership layer needs to be extended with a layer providing a semantics for the messages broadcast to the current group members. This semantics is called *view synchrony* or view-synchronous broadcast (see Section 6.3.6).

- The upper layer provides *atomic broadcast*: it ensures that messages are delivered in the same order by all processes. This dynamic version of atomic broadcast is implemented using the view synchrony layer [BSS91].

[7] The architecture corresponds to the protocol described in [BSS91]. Since we do not discuss *causal order* in the paper, the Isis causal order protocol does not appear here.

Phoenix. The Phoenix architecture [Mal96] is a variation of the Isis architecture (Fig 6.2). The basic layer solves consensus. Membership (primary partition) and view synchrony are provided by the same layer: both the membership problem and view synchrony are solved using the underlying consensus layer. Similarly to the Isis architecture, atomic broadcast is provided on top of the view synchrony/membership layer.

The main limitation of Isis is to provide the membership service at the level of *processors*. In case of partition, this leads the service to kill all processes on processors that are not in the primary partition. This drawback is prevented in Phoenix, which provides the membership service at the level of *processes*. This allows the computation to proceed in all partitions. Consider for example link failures leading to the following situation: the primary partition of some replicated service S is in some network component Π_1, and the primary partition of some other replicated service S' is in some other network component Π_2. A client process in Π_1 can read/update the service S and read S', while a client in Π_2 can read/update S' and read S.

RMP. RMP [WMK94, Mon94] is another monolithic group communication toolkit, whose architecture differs from the Isis and Phoenix architectures (see Figure 6.3). The RMP protocol has been influenced by Chang-Maxemchuk's atomic broadcast algorithm [CM84]. In RMP, the membership layer is split into two parts: *fault-free* membership and *fault-tolerant* membership.

The fault-free membership handles joins and leaves in the absence of failures, using the underlying atomic broadcast layer: joins/leaves are implemented using atomic broadcast. This totally orders joins/leaves with respect to any other application message that is issued using atomic broadcast, i.e., it ensures the *view synchrony* property in the absence of failures. However, the atomic broadcast protocol blocks in case of a process crash. The role of the fault-tolerant membership layer is to avoid blocking by excluding processes that are suspected to have crashed. The fault-tolerant membership protocol, based on a two-phase commit protocol [BHG87] among the surviving processes, is completely different from the fault-free protocol. This fault-tolerant protocol has also the responsibility to ensure the view synchrony property, i.e., it orders view changes with respect to application messages that are atomically broadcast.

Totem. Unlike the architectures presented so far, Totem [AMMS+95] – while being a monolithic architecture – is a representative of the systems based on the *partitionable membership* model.

Similarly to RMP, Totem uses an atomic broadcast algorithm based on a rotating token. Total order is provided by the middle layer of the architecture depicted in Figure 6.4 (the layer handles also flow control). The lower layer membership protocol, apart from detecting failures and defining views, recovers token and messages that had not been received by some members when failures occur. The top

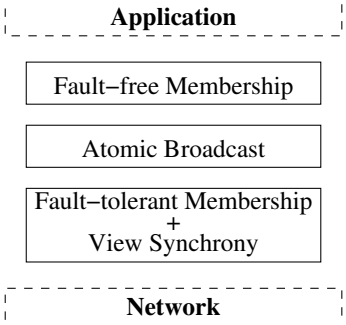


Figure 6.3: RMP architecture

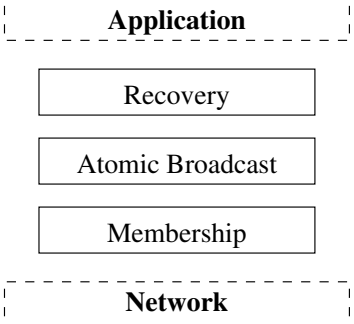


Figure 6.4: Totem architecture

recovery layer completes the membership layer, by ensuring the (extended) view synchrony property.[8] When the membership layer is invoked, e.g., to exclude a process, it does not enforce the (extended) view synchrony property. This is ensured by the recovery layer.

6.4.2 Modular Protocol Stacks

Unlike monolithic systems, modular systems allow users to customize the protocol stack to their specific needs. Horus [vRBG+96] (the successor of Isis) and the re-implementation of Horus in the OCaml language called Ensemble [Hay98] are the best representatives of modular group communication stacks. The idea is to use a set of off-the-shelf components and to compose them using the Horus/Ensemble framework to obtain a protocol stack with the functionalities customized to the user requirements. Similarly to Horus, Ensemble is based on the partitionable membership model. A sample Ensemble protocol stack is depicted in Figure 6.5. A few explanations are needed:

- A component, e.g., *stable*, can be placed at many places in the stack. The choice of the place has an impact on efficiency. For example, the role of the *stable* component is to detect message stability.[9] When stability is detected by the *stable* component, an event is delivered to the layer below, and travels down from layer to layer until it reaches the bottom of the stack. At this point the event is bounced back, and travels up through the stack from component to component, until it reaches the top of the stack. The notification of stability occurs during the *upwards* travel of the event.

- The application is not the uppermost layer in the stack. The reason is that it would take more time to convey events from the network level to the application. The most efficient layering leads placing components active in *normal*

[8] Extended view synchrony [CKV01] extends the view synchrony property, defined in the context of the primary partition model, to the partitionable membership model.

[9] A message is stable at a process when the process knows that the message has been delivered at all destinations.

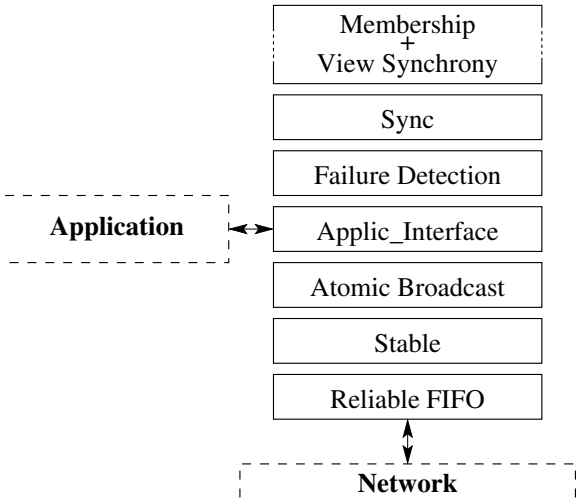


Figure 6.5: Ensemble sample protocol stack

scenarios *below* the application, and components that handle *abnormal* scenarios *above*.

Apart from these generalities, here are comments about the Ensemble stack example depicted in Figure 6.5:

- The *atomic broadcast* component only orders messages in the absence of failures, or more precisely, when the system is stable (since Ensemble provides a partitionable membership service). Without additional membership layers, the different atomic broadcast protocols used would block in case of failures (e.g., upon crash, or disconnection).

- *Sync:* The layer implements a protocol for blocking a group during view changes, i.e., for preventing the broadcast of new messages during view changes.

- *Membership:* Actually, this is not a single layer, but a protocol suite, which includes various components, e.g., *merge*, *inter*, *intra*, etc. It is important to note that even though the membership component appears above the atomic broadcast component, it does not rely on it at all; a correct stack can have the membership components without the atomic broadcast component.

6.5 Roadmap to the Rest of Part II

The remaining two chapters of Part II present our contributions. Chapter 7 presents a new architecture for group communication that overcomes the problems found in

the traditional way to structure group communication protocols in the dynamic crash-stop model. Chapter 8 considers the static crash-recovery model and addresses the specification of atomic broadcast in that model. Additionally, the chapter makes clear the distinction between uniform and non-uniform protocols in the crash-recovery model.

Chapter 7

A New Architecture for Group Communication in the Dynamic Crash-Stop Model

In this chapter, we propose a new architecture for group communication middleware. Traditional group communication toolkits share some common features, despite the big differences that exist among them. We first point out these common features. Then we show the features of our new architecture, which provides several advantages over the existing ones: (1) it is less complex, (2) it defines a set of group communication abstractions that is more consistent than the abstractions usually provided, and (3) it can be made more responsive in case of failures.

7.1 Introduction

In the dynamic group system model, processes are organized into groups. As presented in Section 6.3, the membership of a group can change over time, as processes *join* or *leave* the group, or as crashed processes are *removed* from the group. The current set of processes that are members of a group is called the *group view*. Processes are added to and deleted from the group view via *view changes*, handled by a *membership* protocol. Communication to the members of a group is done by various *broadcast* primitives. The basic "reliable" broadcast primitive in the context of a view is called *view synchronous broadcast*, or simply *view synchrony*. The semantics of view synchronous broadcast can be enhanced by requiring messages to be delivered in the same order by all processes in the view. This primitive is called *atomic broadcast*. Moreover, different research groups distinguish between the *primary partition* membership and *partitionable* membership. The discussion of these two models is outside the scope of this thesis, where we focus on the primary partition model, in which processes observe the same sequence of views. Primary partition membership is adequate for managing replicated servers, even in the case of link failures and/or network partitions (see Sect. 6.2.3).

The first observation we made is that all the implemented group communication toolkits we are aware of, adopt the same basic architecture, in which the group membership and view synchrony protocols are the basic components in the system. The guarantees provided by these two basic components are then used to implement other group communication protocols, e.g., atomic broadcast. We call this architecture the "traditional architecture".

Contribution: a New Architecture. We propose a new architecture with two key features that distinguish it from traditional architectures.

The first key feature is atomic broadcast (instead of group membership and view synchrony) as the basic component. The atomic broadcast component is then used to build other group communication protocols on top, e.g., group membership. Such an architecture has better separation of concerns. For example, the group membership protocol usually has to deliver new group views with guarantees that resemble those provided by the atomic broadcast. Therefore it seems logical for the atomic broadcast protocol to be more primitive than the group membership protocol. This architecture is formally supported by the new specification of group communication for dynamic groups given in [Sch06].

The second key feature of our new architecture is the absence of view-synchronous broadcast. This traditional protocol, which has a rather complex specification (see Section 6.3.6), is replaced by the generic broadcast protocol presented in Sect. 6.3.5. Generic broadcast has a simpler specification than view-synchronous broadcast, but at the same time provides more general guarantees.

In our opinion, the reason for adopting the traditional architecture in the implementation of group communication systems is more historical than justified by some strong arguments. At the time when the first group communication systems (such as Isis) were built, it was not clear how to implement fault-tolerant atomic broadcast protocols without reconfiguration to exclude crashed processes (i.e., using dynamic groups). In later years, when the first papers appeared that suggested a different implementation of atomic broadcast (for example [CT91]), the traditional architecture had been already well established and the new implementations of group communication toolkits usually followed this initial approach.

In this chapter, we argue that our new architecture is not only more elegant than the traditional architecture, but also has several advantages, which make it an interesting choice for designing and implementing new generations of group communication systems. The rest of the chapter is organized as follows. Section 7.2 discusses the common features of traditional architectures. Section 7.3 describes our new architecture. Section 7.4 discusses the advantages of the new architecture compared to the traditional architecture. Finally, Section 7.5 concludes the chapter.

7.2 Discussion on Existing Group Communication Architectures

We identify the following three common features in traditional group communication architectures.

7.2.1 Group Membership and Failure Detection Are Strongly Coupled

Failure detection is a lower level mechanism than *group membership*. Failure detection gives notification of (possible) process failures (or disconnection) without worrying about inconsistencies (e.g., process p might suspect process r, whereas q might never suspect r). On the other hand, group membership gives *consistent* failure notification.[1]

However, none of the existing architectures exploits this difference: group membership and failure detection are strongly coupled. In most of the architectures, the failure detection component does not even appear explicitly: it is completely hidden within the group membership component. Even in the architectures where the failure detection component is not hidden in the group membership, it directly interacts with the group membership, and only with it. In other words, other components learn about suspicions from the group membership component, not from the failure detection component. The group membership component acts as a failure detection component for the rest of the system.

7.2.2 Atomic Broadcast Algorithms Rely on Group Membership

A corollary of the previous observation is that, in all architectures implementing dynamic groups, atomic broadcast algorithms rely on the group membership component; all these algorithms require the help of group membership to avoid blocking in the case of the failure of some critical process.

Basically these atomic broadcast algorithms operate in two modes: (1) a failure-free mode, and (2) a failure mode. A failure notification received from the group membership leads the protocol to switch from the failure-free mode to the failure mode. Here are two examples:

- In Isis and Phoenix, atomic broadcast is implemented using a fixed *sequencer* process. In the normal mode, the sequencer process attaches sequence numbers to messages that are atomically broadcast. However, the protocol blocks if the sequencer crashes. The notification of the failure of the sequencer is needed to prevent blocking, and to switch to the failure mode. In the failure mode the algorithm ensures that if one process has received a sequence

[1] The notion of *consistency* differs in the primary and in the partitionable membership specifications.

number for some message m, then all correct processes receive the same sequence number for m. Once this is ensured, a new sequencer is chosen, and the algorithm returns to the normal mode.

- In RMP and Totem, processes form a logical ring and atomic broadcast is implemented using a rotating *token*. In the normal mode, the token is passed over the ring of processes. A process holding the token can attach a sequence number to the messages it wants to broadcast. If one process crashes, the ring is broken, and the token may be lost. The failure mode is needed to recover from this situation.

This dependency of atomic broadcast on group membership is visible in the protocol stacks, where the membership component is *below* the atomic broadcast component. This is only partially true for RMP (see Figure 6.3), in which the dependency of atomic broadcast on group membership holds only in case of failures (failure-free membership is implemented using atomic broadcast). This dependency of atomic broadcast on group membership also holds in Ensemble, even though the atomic broadcast component is below the membership component in the stack in Figure 6.5: in Ensemble the layering of components does not reflect functional dependencies.

7.2.3 The Consensus Abstraction Is Barely Used

When the consensus problem was defined in the early eighties [Fis83], it was largely considered as a theoretical problem, with little practical relevance. Since then, the practical importance of consensus for solving problems such as atomic broadcast, (primary partition) group membership or view synchrony has been recognized. Nevertheless, except for Phoenix, no consensus component appears in the implementations.

Notice that this comment about consensus applies only to the primary partition systems, since the role of consensus in the context of partitionable group membership and extended view synchrony [CKV01] (the counterpart of view synchrony in the context of partitionable membership) is not clear.

7.3 The New Architecture

We present now our new architecture. We proceed in three steps: we start with an overview; then in Section 7.3.2, we present the augmented version of the architecture with a new key component: *generic broadcast*. Finally in Section 7.3.3, we describe the full version of the architecture with additional details.

7.3.1 Overview of the New Architecture

Figure 7.1 shows an overview of our new architecture. At this level of details, we can already see three important features:[2]

- Atomic broadcast does not rely on group membership, but *group membership relies on atomic broadcast*.
- There is no *view synchrony* component.
- Group membership and failure detection are decoupled.

Group Membership Relies on Atomic Broadcast and Not the Opposite. Traditional group communication toolkits rely on atomic broadcast algorithms that require a perfect failure detector, i.e., a failure detector that makes no mistakes. This failure detector is denoted by $\mathcal{P}$ (see Sect. 6.3.2). The group membership protocol, when placed below atomic broadcast, emulates the perfect failure detector $\mathcal{P}$ by forcing incorrectly suspected processes to crash.

Instead, we can use an atomic broadcast algorithm requiring a $\diamond\mathcal{S}$ failure detector (much weaker than $\mathcal{P}$), which allows to make mistakes by suspecting correct processes: $\diamond\mathcal{S}$ allows even an unbounded number of wrong suspicions. Such an atomic broadcast algorithm is given in [CT96]: it is based on a sequence of instances of consensus (see the *consensus* component in Figure 7.1 below the total order broadcast component). This algorithm is able to work without blocking even if up to $f < n/2$ crashes occur. As a result, this algorithm does not have to rely on a group membership protocol.

Since the group membership component does not need to appear *below* the atomic broadcast component, it can be placed *above*: this means that group membership can be implemented using atomic broadcast, which is quite natural, since views need to be totally ordered. This generalizes RMP's solution (see Sect. 6.4.1). However, because of the limitations of the atomic broadcast algorithm used by RMP (it assumes a perfect failure detector, emulated by the membership protocol), RMP could use the solution only in the absence of failures: RMP's atomic broadcast relies on membership in case of failures.

It might appear to the reader that inverting the group membership component and the atomic broadcast component in the stack is just moving the complexity from one component to the other (the more complex component being the lowest in the stack). This is not true. It should be noted that any solution that implements (primary partition) group membership *below* atomic broadcast, actually has two algorithms to solve the same ordering problem: one specific solution to order membership changes, and one general (in the context of atomic broadcast) to order application messages. This only observation suggests that such architectures are not optimal.

[2] Note that Figure 7.1 does not mean that the application can only interact with the Group Membership component (the component just below the application).

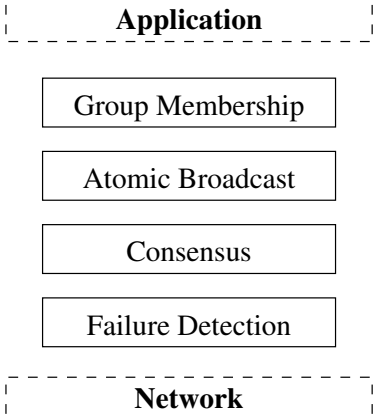


Figure 7.1: New architecture: overview

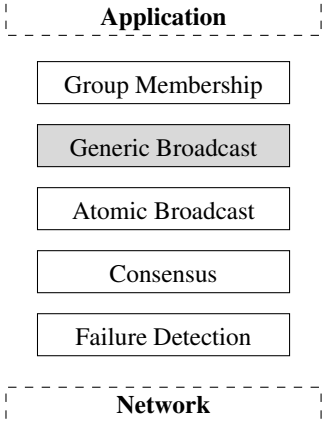


Figure 7.2: New architecture with the Generic Broadcast component

There Is No View Synchrony Component. There is no view synchrony component in Figure 7.1. This component is replaced by a more powerful component, called *generic broadcast*, discussed below.

Group Membership and Failure Detection Are Decoupled. The strong coupling between failure detection and group membership in traditional architectures was motivated by the atomic broadcast algorithms (requirement of a perfect failure detector emulated by the membership protocol). These architectures could not exploit the distinction between failure suspicion and membership exclusion (only process exclusions could be exploited by the atomic broadcast algorithm).

Decoupling group membership from failure detection has the following advantage: failure detections do not necessarily lead to process exclusion. This also means that decisions to exclude processes are no more taken by the group membership component. We come back to this issue below.

7.3.2 Augmented Version of the New Architecture

We introduce now the key component of our new architecture, namely *generic broadcast* (see Figure 7.2).

Generic Broadcast Component. Generic broadcast is a powerful group communication primitive presented in Sect. 6.3.5. It is generic in the sense that the ordering of messages is defined by a conflict relation on the messages. If two *conflicting* messages m and m' are broadcast, then generic broadcast delivers them in the same order on all destination processes. However, if m and m' do *not* conflict, then generic broadcast does not order them (which is less expensive).

In terms of the implementation of our architecture, we assume here a thrifty implementation of generic broadcast that uses atomic broadcast [ADGFT00]. In such

a solution, atomic broadcast is not necessarily called in every run. Atomic broadcast is used only when conflicting messages are broadcast (see [ADGFT00] for an extended discussion of the notion of *thrifty* implementation of generic broadcast).

Active and Passive Replication. Since a group communication middleware is supposed to provide abstractions for the replication of critical components, it is natural to confront the abstractions provided so far with the needs of replication techniques. Our preliminary architecture (Fig. 7.1) provides atomic broadcast, which allows us to implement active replication [Sch93], also called state machine approach (in active replication, the client requests are sent to all servers using atomic broadcast, every server processes the request, and sends the response to the client).

Atomic broadcast is not needed in passive replication. Instead, view synchrony provides the right abstraction, see for example [GS97]. However, our new stack does not provide such an abstraction. We illustrate in the next section how generic broadcast can be used in place of view synchrony. More generally, as shown in [Sch06], view synchrony does not need to be considered as a basic abstraction. View synchrony follows rather from adequate specifications of dynamic group communication [Sch06].

Generic Broadcast instead of View Synchrony for Passive Replication. In passive replication, the client sends its request to only one server, the *primary*. Only the primary processes the client request; before sending the response back to the client, the primary updates the state of the backups. This is done by an *update* message, sent from the primary to the backups. The standard solution consists in relying here on *view synchrony*.

With generic broadcast,[3] the solution consists in considering two types of messages (Fig. 7.3): (1) *update* messages, and (2) *primary change* messages. The *update* messages are used by the primary to update the state of the backups. The *primary change* messages are used by the backups to change the new primary, when the current primary is suspected to have crashed. A *primary change* message does not lead to the exclusion of the old primary, which remains in the view. If the primary has actually crashed, a new view will be installed to exclude it after a very long timeout (see *Monitoring Component*, Section 7.3.3).

The conflict relation between *update* and *primary change* messages is as follows:

	update	primary change
update	*no conflict*	*conflict*
primary change	*conflict*	*conflict*

[3] In this example we have to assume FIFO generic broadcast, i.e., the FIFO point-to-point property in addition to the ordering properties of generic broadcast. The same FIFO property is required in the context of the solution based on view synchrony.

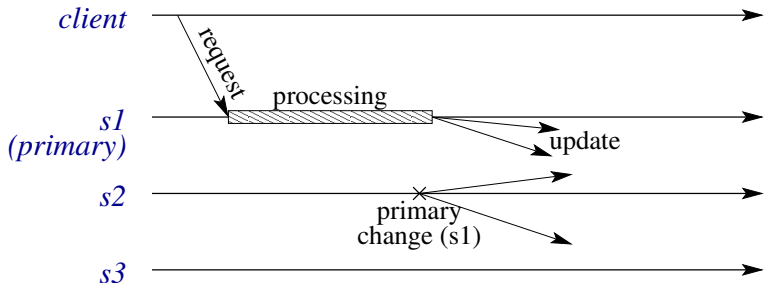


Figure 7.3: Generic broadcast for passive replication

This conflict relation ensures that (1) *primary change* messages are totally ordered, (2) *update* messages are totally ordered with respect to *primary change*, and (3) *update* messages are not ordered with respect to other *update* messages.[4] For illustration, consider a replicated server with three replicas s_1, s_2, s_3 (which define the group s) and the following scenario (Figure 7.3):

- The server s_1 is initially the primary.
- At time t, s_1 receives a client request, processes it, and generic-broadcasts the *update* message to the group s.
- Approximately at the the same time t, server s_2 suspects s_1 to have crashed, and generic-broadcasts the *"primary-change(s_1)"* message to the group s. Upon delivery of this message, all servers (including s_2) modify their view from $[s_1; s_2; s_3]$ to $[s_2; s_3; s_1]$, which leads the servers to consider s_2 to be the new primary.[5]

Since these two messages conflict, we have only two possible outcomes:

1. All members of s deliver the *update* message before the *primary-change* message.
2. All members of s deliver the *primary-change* message before the *update* message.

In case 1, the primary change occurs logically *after* the handling of request req by s_1. In case 2, the primary change occurs logically *before* the handling of the request. This means that the processing of the request by s_1 must be ignored. The client will timeout, learn that s_2 is the new primary, and reissue its request to s_2.

[4] A symmetric conflict relation could also be considered: all messages conflict except *primary change* messages among them.

[5] Views are here *lists* of processes, rather than *sets* of processes. The primary is the process at the head of the list. Note that the delivery of the *primary-change(s_1)* message does not lead to the exclusion of s_1.

7.3.3 Full Version of the New Architecture

The full version of our architecture, which includes all components and all interfaces between components, is given in Figure 7.4. The additional components are:

- the *reliable channel* component,
- the *monitoring* component.

Note that in Figure 7.4, the operations on the generic broadcast component are called *abcast* (invocation of atomic broadcast) and *rbcast* (invocation of reliable broadcast).[6] The conflict relation is the following:

	rbcast	abcast
rbcast	*no conflict*	*conflict*
abcast	*conflict*	*conflict*

In other words, in the context of the passive replication example, *rbcast* should be used for the *"udpate"* message, and *abcast* for the *"new primary"* message. Of course, generic broadcast can be initialized with a different conflict relation table.

We explain now briefly the role of the *reliable channel* and *monitoring* components.

Reliable Channel Component. This component ensures the set of properties defined in Sect. 6.3.1 for quasi-reliable channels. This abstraction can be efficiently implemented on top of TCP [EUS02].

Monitoring Component. In our architecture, the decision to exclude a suspected process from the membership is not made by the group membership component.[7] The decision is made by the monitoring component, which then calls the *remove* operation of the *membership* component.

The separation of concerns between the *failure detection* component and the *monitoring* component allows for very flexible policies. On the one hand, the *consensus* component of process p could ask the *failure detection* component to use a *small* timeout value (e.g., in the order of seconds) to suspect some other process q in order to avoid blocking for a long time. Typically, this suspicion would *not* lead to the exclusion of q. On the other hand, the *monitoring* component of p might ask the *failure detection* component to use a large timeout value (e.g., in the order of minutes) to suspect q. Here a suspicion would lead the *monitoring* component

[6] See [Sch06] for a precise specification.

[7] The operations on the membership component are *join* – to add a process to the group, and *remove* – to remove a process from the group (including itself).

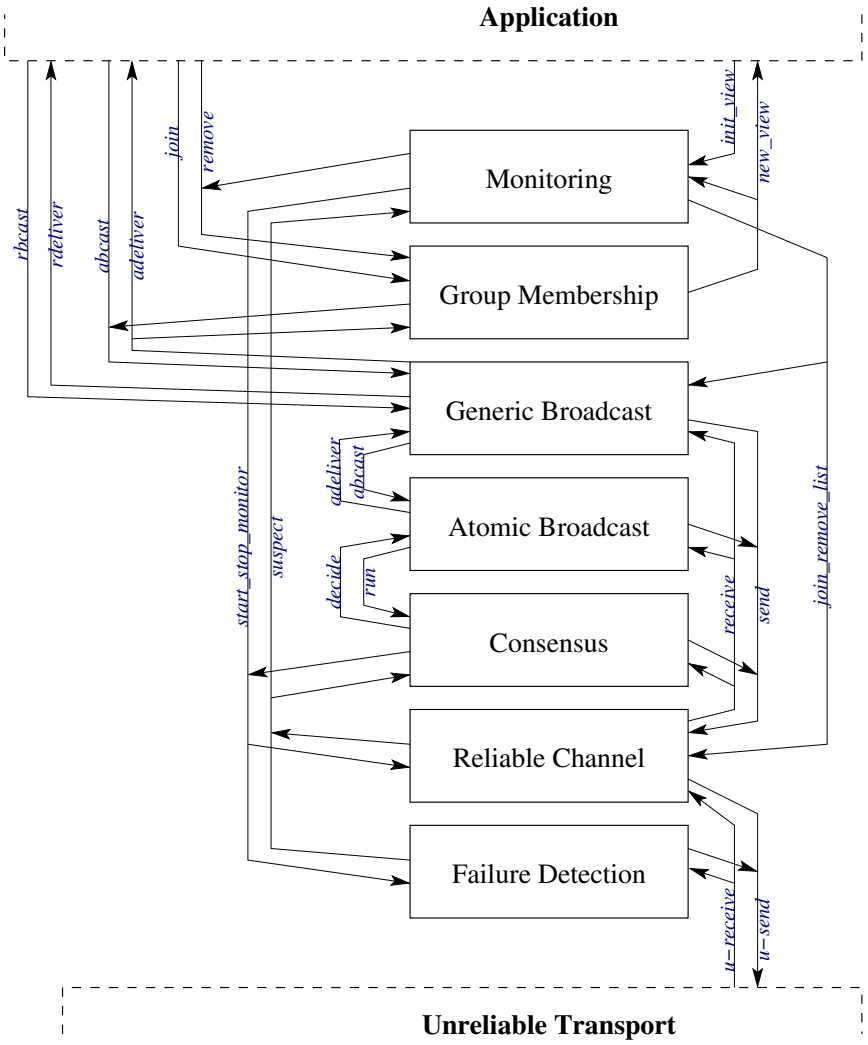


Figure 7.4: New architecture: full version

to call the *membership* component to *remove* q. However, to make such a decision, the monitoring component may also interact with the *monitoring* component of other processes, and for example decide on the removal of q only after having learned that a threshold of other processes also suspect q.

Still another exclusion policy can be expressed, which is relevant when process p sends message m to process q. The *reliable channel* component at p buffers m, until $ack(m)$ is received from q (which acknowledges reception of m by q). If q crashes, m might stay in p's buffer forever. In this case, the only way to discard m is to exclude q from the membership (if q is excluded from the membership, there is no more obligation for q to deliver m, i.e., m can be safely discarded). This is called *output-triggered* suspicion in [CBDS02]. The *monitoring* component can

exclude processes based on output-triggered suspicions (which are only necessary when the output buffers are almost full).

7.4 Assessment of the New Architecture

We stress now on the advantages of the new architecture compared to the traditional architectures.

7.4.1 Less Complex

In traditional architectures, the ordering problem is solved in two places: (1) within the group membership component for views, and (2) within the atomic broadcast component for messages.[8] From a conceptual point of view this is not optimal, and introduces an unnecessary complexity. This redundancy has disappeared in the new architecture, where the ordering problem is solved only once (in the atomic broadcast component).

Actually, in the traditional architectures the ordering problem is even solved in a third place, namely in the view synchrony component, which orders messages with respect to view changes. In our new architecture, this additional ordering problem is also solved in the same place, namely in the atomic broadcast component. Indeed, when the generic broadcast component detects a message conflict (e.g., between a reliable broadcast and atomic broadcast view change message), then it calls the atomic broadcast component. The details can be found in the thrifty generic broadcast algorithm [ADGFT00].

Altogether, from the point of view of the ordering problem, the new architecture is less complex than traditional architectures. Smaller complexity usually leads to easier maintenance.

7.4.2 More Powerful (Provides More Functionalities)

The new suite of components provides functionalities which are not present in traditional stacks. The prominent example is generic broadcast, which extends the ordering provided by view synchrony. Consider for example a replicated service managing client bank accounts, with deposit and withdrawal operations (withdrawal does not allow to withdraw more than available). Both classes of operations update the state of the server, but deposit operations are commutative, i.e., they do not need to be ordered with respect to themselves. This ordering typically can be solved using generic broadcast. Traditional stacks do not provide any specific solution: atomic broadcast would have to be used both for deposit and withdrawal operations. This would induce a non-necessary overhead.

On a more minor issue, the fact that failure suspicions can be generated in two distinct places is not without benefit. Depending on the context, the monitoring

[8] In RMP, ordering is performed in two different places only in case of failures.

component can take the decision to exclude a process from the membership either (1) based on notification from the failure detector component, or (2) based on notifications from the reliable channel components, or (3) it could wait for notifications from both components.

7.4.3 Higher Responsiveness

Group communication allows the implementation of fault-tolerant replicated services. Performance of group communication is usually measured in failure-free executions. However, performance of group communication in case of failures often is equally important.

Consider for example the latency of atomic broadcast, i.e., the time elapsed between the atomic broadcast of m and the first delivery of m. In case of failures, the timeout used to detect failures represents an important part of this latency. So, reducing the latency in case of failures requires failure detection timeouts to be as small as possible. However, reducing failure detection timeouts increases the probability of false suspicions. Decoupling failure suspicions from process exclusions plays here an important role.

In traditional architectures, wrong failure suspicions have a high cost: the cost of excluding the wrongly suspected processes, followed by the cost of the join operation (with the costly state transfer operation) in order to include again the process in the membership. This has forced traditional systems to adopt large failure detection timeout values. In our stack, where failure suspicions are decoupled from exclusions (i.e., false suspicions lead to a small overhead), timeouts can be chosen to be smaller. This leads to a gain in efficiency in case of failures, e.g., to higher responsiveness.

7.4.4 Minor Efficiency Issue

Traditional systems have another responsiveness problem, namely in the context of view changes. This problem is not related to failures, since view changes may be triggered by *join* requests, and *remove* requests that are not exclusions. The traditional solution in the context of membership changes ensures that messages broadcast before the membership change are delivered before the membership change takes place. This property is defined as *sending view delivery* (see Sect. 6.3.6). However, in order to ensure this property without discarding messages, processes must stop sending messages while the membership change protocol is running (see for example the *Sync* layer of Ensemble, Section 6.4.2). To prevent this undesirable *blocking* problem, which reduces responsiveness, alternate and more complex solutions to handle membership changes have been proposed. These solutions implement a weaker property called *same view delivery* (see Sect. 6.3.6). The implementation based on generic broadcast does not lead to blocking: the solution "naturally" implements the *same view delivery* property without additional complexity [Sch06].

7.5 Conclusion

Existing group communication systems (GCS) can be classified according to two dimensions: (1) the *membership model* dimension, and (2) the *structuring* dimension. The *membership model* dimension allows the classification of GCS as either (i) *primary partition* GCS, or (ii) *partitionable membership* GCS. The *structuring* dimension allows the classification of GCS as either (i) *monolithic* or (ii) *modular*. Isis, falls into the category *primary partition/monolithic*, while Ensemble falls into the category *partitionable/modular*.

This chapter has introduced a third dimension: the *protocol* dimension. With respect to this third dimension, existing GCS can be characterized as *GM-VS*:[9] (1) membership is the basic component in the stack, and (2) view synchrony is the basic communication abstraction. The chapter has presented an alternate solution that could be called *AB-GB*[10] based: (1) atomic broadcast is the basic component, (2) no view synchrony as such is provided, and (3) the GCS provides generic broadcast (instead of view synchrony) as a more powerful abstraction.

The Fortika group communication toolkit (see Chapter 9) has been implemented following this new architecture. Fortika is described in Chapter 9.

[9] Group Membership - View Synchrony.
[10] Atomic Broadcast - Generic Broadcast.

Chapter 8

Improving Atomic Broadcast in the Crash-Recovery Model

Most of the research work devoted to group communication protocols and to atomic broadcast in particular has been done in the context of the crash-stop model. The drawback of this model is its inability to express algorithms that tolerate the crash of a majority of processes. The problem is worse in the static model: crashed processes stay crashed and can not be replaced by new ones. This has led to extend the crash-stop model to the crash-recovery model, in which processes have access to stable storage, to log their state periodically. This allows them to recover a previous state after a crash.

However, the existing specifications of atomic broadcast in the crash-recovery model are not satisfactory. In this chapter, we propose a new specification of atomic broadcast in the crash-recovery model that addresses the issues found in previous specifications. Specifically, our new specification allows us to distinguish between a uniform and a non-uniform version of atomic broadcast. The non-uniform version logs less information, and is thus more efficient. Performance results are presented.

8.1 Introduction

Atomic broadcast ensures that messages broadcast by different processes are delivered by all destination processes in the same order (see Sect. 6.3.5). Many atomic broadcast algorithms have been published in the last twenty years [DSU04]. Almost all of these algorithms have been developed in a model where processes do not have access to stable storage: the *crash-stop* model (see Sect. 6.3.5). In such a model, a process that crashes loses all its state; upon recovery it cannot be distinguished from a newly starting process. The crash-stop model is attractive from an efficiency point of view: since logging to stable storage is a costly operation, atomic broadcast algorithms that do not log any information are significantly more efficient than atomic broadcast algorithms that access stable storage.

However, atomic broadcast algorithms in the crash-stop model also have drawbacks: they tolerate only the crash of a minority of processes. Moreover, there are contexts where access to stable storage is natural, e.g., database systems. It has been shown that replicated database systems can benefit from atomic broadcast [AAEAS97, PGS98, WS05, KA00, PMJPKA00, Kem00, Wie02, AT02] but atomic broadcast in the crash-stop model does not suit this context [WS04].

For this reason, there is a strong motivation to consider atomic broadcast in the crash-recovery model, where processes have access to stable storage to save part of their state: a process that recovers after a crash can retrieve its latest saved state, and restart computation from there on. Because of the strong link between consensus and atomic broadcast (if one problem is solvable, the other is also solvable), the more basic of these two problems, namely consensus, needs to be addressed first. Among the papers that address crash-recovery consensus [OGS97, HMR98, ACT00], we highlight the work by Aguilera *et al* [ACT00]. They define a new failure detector for the crash-recovery model and propose two algorithms for solving consensus in that model. Based on this result, Rodrigues and Raynal address the problem of atomic broadcast in the crash-recovery model [RR03]. While this paper advances the state-of-the-art, it has some weaknesses. From our point of view, the main problem in [RR03] is the *specification* of atomic broadcast. As we have seen in Sect. 6.3.5, the classical specification of atomic broadcast is in terms of the two primitives *abcast* and *adeliver*. No *adeliver* primitive appears in [RR03], where *adeliver* is a predicate. The value *true* / *false* of the predicate depends on a sequence of messages called *adeliver-sequence*. The application has to poll this sequence for newly adelivered messages. This shows a problem in the specification. A more important implication is that the specification in [RR03] does not reduce to the classical specification of atomic broadcast in the crash-stop model when crashed processes do not recover.

We point out another limitation of the work in [RR03]. The work only addresses *uniform* atomic broadcast. *Non-uniform* atomic broadcast in the crash-recovery model is an alternative that can be very interesting from a practical point of view. Non-uniform atomic broadcast can be seen as an intermediate solution, between (1) an atomic broadcast algorithm in the crash-stop model that does not access stable storage at all, and (2) a uniform atomic broadcast algorithm in the crash-recovery model that is expensive due to frequent accesses to stable storage. In contrast to [RR03], we propose both a *uniform* and a *non-uniform* version of atomic broadcast. The non-uniform atomic broadcast algorithm does not require frequent access to the stable storage. Interestingly, our two specifications reduce to the classical specification of atomic broadcast in the crash-stop model when crashed processes do not recover.

We also explain why atomic broadcast in the crash-recovery model is trickier than in the crash-stop model. Atomic broadcast is most of the time used within an application that has a *state*. The atomic broadcast algorithm (and the whole composition that contains it) also has a *state*. Upon recovery both states must be consistent. However, with no recovery this is not a problem! This becomes

a problem in the crash-recovery model. We show how the consistency issue is addressed both in our specification and in our implementation. Finally, we have run experiments that show the gain in performance of the non-uniform version of our atomic broadcast algorithm with respect to the uniform version.

The rest of the chapter is organized as follows. Section 8.2 is devoted to the specification of uniform and non-uniform atomic broadcast in the crash-recovery model. Section 8.3 discusses the problem of keeping the application state consistent with the state of the atomic broadcast algorithm. Section 8.4 presents the two algorithms that satisfy our uniform and non-uniform specification of atomic broadcast. Section 8.5 is devoted to performance evaluation. Finally, Section 8.6 concludes the chapter.

8.2 Specification of Atomic Broadcast in the Crash-Recovery Model

8.2.1 Definitions

In Chapter 6, we defined atomic broadcast with primitives *abcast* and *adeliver*. To these two primitives, we add a third *commit* primitive. Roughly speaking, the commit primitive executed by q marks the point at which q's execution will resume after a crash. When commit is executed by q, all messages previously adelivered at q will never be adelivered again at q, even if q crashes and recovers. The commit primitive addresses the fundamental process state problem in the crash-recovery model. The state of each process q is split into two parts: (1) the application state, and (2) the atomic broadcast protocol state. The distinction between these two states can be ignored when processes do not recover after a crash, but not here. With the commit primitive, we introduce the following terminology:

1. Process q *ab-commits* message m if (1) q abcasts m, (2) q executes the primitive *commit()* later on, and (3) q does not crash in-between.
2. Process q *del-commits* message m if (1) q adelivers m, (2) q executes the primitive *commit()* later on, and (3) q does not crash in-between.

We also introduce the notion of *permanent* and *volatile* event. In our model, events are *crash*, *recover*, and the execution of the primitives presented above. If process q crashes, its volatile events are those whose effect may be lost; q's permanent events are those whose effect is not lost even if q crashes. So if q never crashes, the effect of its whole history is permanent. More formally, the set V_q of volatile events and the set P_q of permanent events partition q's history, denoted by h_q. An event $e \in h_q$ belongs to V_q if (1) a crash event e_c occurs after e in h_q, and (2) no *commit* event occurs in h_q between e and e_c. An event $e' \in h_q$ belongs to P_q if it does not belong to V_q.
We introduce some additional definitions that will be used in the specification of atomic broadcast:

- *Non-Recovery Runs:* Let R_Π be the set of all possible runs allowed in the crash-recovery model for process set Π. We define *non-recovery runs* to be the set $N_\Pi \subset R_\Pi$ of runs that do not contain any commit event or any recover event other than at system start-up time. N_Π is the set of all possible runs in the well-known crash-stop model for process set Π.
- *Permanent Broadcast:* We say that process q *permanently abcasts* (or simply q *p-abcasts*) message m if (a) q ab-commits m, or (b) q abcasts m and does not crash later. In other words, q *permanently abcasts* message m if event *abcast*(m) belongs to set P_q of q's permanent events.
- *Permanent Delivery:* Likewise, we say that process q *permanently adelivers* (or simply q *p-adelivers*) message m if (a) q del-commits m, or (b) q adelivers m and does not crash later. In other words, q *permanently adelivers* message m if event *adeliver*(m) belongs to set P_q of q's permanent events.
- *Delivery Order:* We say that process q *adelivers message* m *before* m' if (a) q adelivers m and later m' and does not crash in-between, or (b) q adelivers m' after having del-commited m. Notation: $m \triangleright_q m'$.

  Note that, if q crashes between the adelivery of m and m', these two messages may not be ordered.
- *Permanent Delivery Order:* We say that process q *p-adelivers message* m *before* m' if (1) $m \triangleright_q m'$ holds, and (2) q p-adelivers m and m'. Notation: $m \triangleright \triangleright_q m'$.
- *Multiple Delivery:* We say that process q *adelivers message* m *more than once* if we have $m \triangleright_q m$.

  Note that, if q adelivers m twice but crashes in-between, then m is not necessarily considered as adelivered more than once.

8.2.2 Specification of Atomic Broadcast

We can now formally define atomic broadcast. As in the crash-stop model, we distinguish between *uniform* and *non-uniform* atomic broadcast. A first attempt to introduce this distinction was made in [BG00] in the context of Reliable Broadcast (which is weaker than Atomic Broadcast since it does not enforce any order on message delivery), but the lack of a primitive like *commit* does not lead to a convincing specification. In our specification, uniform atomic broadcast constrains the behavior of good and bad processes, while non-uniform atomic broadcast does not impose any constraints on (1) bad processes, and (2) volatile events of good processes. In other words, non-uniform atomic broadcast ignores volatile events, and considers only permanent events, i.e., the events that good processes *"remember"* once they stop crashing.

An important feature of our specification of uniform and non-uniform atomic broadcast is that, in non-recovery runs (see Sect. 8.2.1), our new specification reduces exactly to the classical definition of uniform and non-uniform atomic broadcast [HT94]. We define (non-uniform) atomic broadcast by the properties Validity

(1), Uniform Integrity[1] (2), Agreement (4), and Total Order (6) defined below. We define uniform atomic broadcast by the properties Validity (1), Uniform Integrity (2), Uniform Agreement (3), and Uniform Total Order (5).

1. *Validity*: *If a good process q p-abcasts m then q p-adelivers m.*

   There is no uniform Validity property, since it does not make sense to require from a bad process, which can crash and never recover, to deliver m. So the Validity property is the same for uniform and non-uniform atomic broadcast.

2. *Uniform Integrity*: *For every message m, every process q adelivers m only if some process has abroadcast m. Moreover, $m \triangleright_q m$ never holds for any process q.*

   This property allows a process to adeliver the same message twice (under certain conditions), unlike Uniform Integrity in the crash-stop model (see the definition of *multiple delivery*, Sect. 8.2.1). For instance, if process q adelivers message m and then crashes before del-committing m, Uniform Integrity allows q to adeliver m again after recovery.

3. *Uniform Agreement*: *If a process (good or bad) adelivers message m, then every good process p-adelivers m.*

   This property requires that all good processes permanently adeliver any message that is adelivered by some process. The required permanent delivery of m ensures that a good process q *"remembers"* having adelivered m at the time q stops crashing.

4. *Agreement*: *If a good process p-adelivers message m, then every good process p-adelivers m.*

   Non-uniform Agreement only puts a constraint on messages p-adelivered by good processes. There is no constraint on a message adelivered (but not p-adelivered) by a process that later crashes. Also, there is no constraint on a message p-adelivered by a bad process. In the two cases, no process *"remembers"* m.

5. *Uniform Total Order*: *Let p and q be two processes (good or bad). If $m \triangleright_p m'$ holds and q adelivers m', then $m \triangleright_q m'$ also holds.*

6. *Total Order*: *Let p and q be two good processes. If $m \triangleright \triangleright_p m'$ holds and q p-adelivers m', then $m \triangleright \triangleright_q m'$ also holds.*

The introduction of the *commit* primitive is fundamental in our specification. It allows us to distinguish between volatile and permanent events. With this distinction it is fairly easy to define uniformity and non-uniformity in the crash-recovery

[1] We do not define a Non-uniform Integrity property, which does not make much sense from a practical point of view.

model (and it would be hard to introduce the distinction without the *commit* primitive). In addition, with the *commit* primitive, it is *not* the implementor of atomic broadcast who decides when to make events permanent. This is left to the application, which knows better when volatile events are no more interesting (e.g., because an application checkpoint was taken) and should thus become permanent. Moreover, it is easy to see that, if crashed processes never recover, then our specification of uniform and non-uniform atomic broadcast corresponds exactly to the standard specification of uniform and non-uniform atomic broadcast in the crash-stop model.

The non-uniform specification can be criticized with the argument that a bad process p (e.g., a process that crashes and recovers infinitely often) can behave arbitrarily, even if it executes *commit* a number of times. However, p cannot know whether it is good or bad because it may recover in the future and stay up forever. This is similar to the crash-stop model, where a process that crashes in the future is faulty and can thus behave arbitrarily. The practical relevance of non-uniformity is discussed in Section 8.4.3.

8.2.3 Related Work

Atomic broadcast has been specified in the crash-recovery model by Rodrigues and Raynal [RR03]. They define the primitive *abroadcast*(m) (abroadcast of m) and the sequence μ_p = *adeliver-sequence()*. Moreover, *adeliver*(m) is a predicate that is true iff $m \in$ *adeliver-sequence()* at p. Atomic broadcast is then specified by the following properties:

- *Validity:* If a process adelivers a message m, then some process has abroadcast m.
- *Integrity:* Let μ_p be the delivery sequence at process p. Any message appears at most once in μ_p.
- *Termination:* For any message m, (1) if the process that issues *abroadcast*(m) returns from *abroadcast*(m) and is a good process, or (2) if a process *adelivers* message m, then all good processes *adeliver* m
- *Total Order:* Let μ_p = *adeliver-sequence()* at process p. For any pair of processes (p, q), either μ_p is a prefix of μ_q or vice-versa.

This specification has several problems. The main one is the absence of an *adeliver* primitive: how is the adeliver-sequence defined? This is the tricky issue that is not addressed. In the group communication literature, specifications usually define the adeliver primitive first, and then the adeliver-sequence as the sequence of messages adelivered. It is the opposite that is done: adeliver is defined based on the adeliver-sequence, and the adeliver-sequence is only defined with the properties of atomic broadcast, thus creating a circularity in the definitions. Moreover, because of the absence of an adeliver primitive, the specification does not reduce to the standard specification of atomic broadcast in the crash-stop model. As a result, all properties derived from the crash-stop model have to be reinvented.

The authors of [RR03] also mention the following problem with their Termination property. If the call to *abroadcast*(m) returns at a good process p, this forces all good processes to eventually adeliver m. They argue that this is problematic to ensure if the process crashes shortly after having called *abroadcast*(m). In contrast, our specification uses the commit primitive, which avoids the problem. Our Validity property only forces good processes to eventually adeliver message m if p *permanently* abcasts m (e.g., p abcasts m and then executes *commit*).

Finally, Rodrigues and Raynal also propose an optimized implementation of atomic broadcast. In this implementation, atomic broadcast checkpoints the state of the application from time to time. We claim that, usually, these checkpoints should be initiated by the application (which knows best when to checkpoint its own state), but this can not be done in the implementation given in [RR03]. The reason is that the specification lacks a primitive (like commit) that the application could use to initiate such a checkpoint.

8.3 Keeping the Process State Consistent

Atomic broadcast is commonly used to update the state of replicated servers. Consider a replica p_i. The state of p_i needs to be distinguished from the state of the atomic broadcast composition local to p_i. We introduce the following notation: s_i^{appl} denotes the application state of p_i and s_i^{abcast} denotes the state of the atomic broadcast composition local to p_i. We assume here that s_i^{appl} and s_i^{abcast} are part of the same OS process denoted by p_i. The distinction between the s_i^{abcast} state and the s_i^{appl} state of p_i can be completely ignored in the crash-stop model. This is no more the case in the crash-recovery model, where p_i must recover in a state such that s_i^{abcast} and s_i^{appl} are consistent. We now address this problem.

8.3.1 Usage of *commit*

We extend the notation just introduced to denote by p_i^{appl} the application *code* of p_i, and by p_i^{abcast} the atomic broadcast *code* of p_i. In order to recover the state s_i^{appl} after a crash, p_i^{appl} checkpoints s_i^{appl} from time to time. After a crash, p_i^{appl} recovers in the most recently saved s_i^{appl} state. From the point of view of p_i^{appl}, the message delivery sequence should resume exactly where it was at the moment of the checkpoint: the delivery (1) must not include any message logically included in s_i^{appl} (a message is logically included in the checkpointed state if it led to the update of s_i^{appl}), but (2) must not miss any message adelivered later in the logical adelivery sequence. For example, consider the logical adelivery sequence m_1, m_2, m_3. If p_i^{appl} has checkpointed its state after the adelivery of m_1 and crashed after the handling of m_2, then the delivery after recovery should restart with m_2. The *commit* primitive fits this requirement naturally: process p_i^{appl} checkpoints s_i^{appl} and then immediately executes *commit*. Condition (1) above is guaranteed by the (Uniform)

Integrity property (which ensures that no del-commited message will be adelivered again); condition (2) is ensured by the Agreement property.

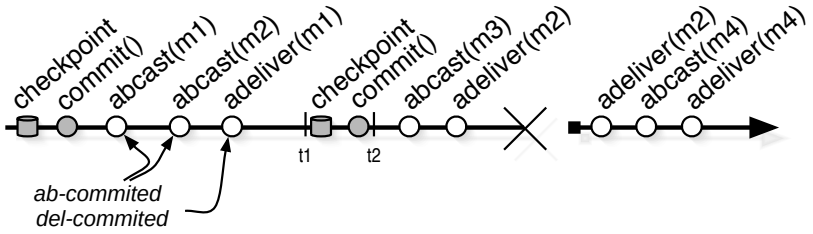


Figure 8.1: Example execution. After p_i^{appl} checkpoints its state, it calls *commit*.

This solution works as long as the checkpoint and the commit operations are executed atomically, that is, a process can never crash between $t1$ and $t2$ in Figure 8.1. Moreover, all events (depicted as circles in Fig. 8.1) are assumed to be atomic so far. We now explain how these two assumptions can be relaxed, while keeping s_i^{appl} and s_i^{abcast} consistent upon recovery.

8.3.2 Addressing the Atomicity Problem

Since s_i^{appl} and s_i^{abcast} are in the same OS process, an obvious and simple way to keep them consistent is a checkpoint mechanism (triggered by a call to *commit*) that atomically checkpoints s_i^{appl} and s_i^{abcast}. However, this solution introduces a dependency between the application and the atomic broadcast protocol. In this section, we present a simple solution that keeps the atomic broadcast protocol and the application independent of each other.

For a process p_i, we have introduced the distinction between p_i^{appl} and p_i^{abcast}. The interaction between p_i^{appl} and p_i^{abcast} is naturally expressed by means of function calls (e.g., *abcast* function, *commit* function). Function calls are synchronous: the caller blocks while the call is being executed. This yields a useful property: the caller is sure that the callee has completely processed the call when it returns. Consider now that p_i crashes during the function call (e.g., during *abcast* or *commit*). When p_i recovers, it does not know whether the function was successfully executed or not.

To address this problem, we model the communication between p_i^{appl} and p_i^{abcast} in terms of *messages*. When p_i^{appl} invokes primitive F(PARAMETERS) on the atomic broadcast interface, we say it sends the (local) message $< F$, PARAMETERS$>$ to p_i^{abcast} (see Fig. 8.2). Likewise, when p_i^{abcast} invokes primitive F'(PARAMETERS') on the application interface, we say it sends the (local) message $< F'$, PARAMETERS'$>$ to p_i^{appl} [WS04].

When modeling intra-process communication using the message-passing model, a single process p_i becomes a *distributed system* with two processes p_i^{appl} and

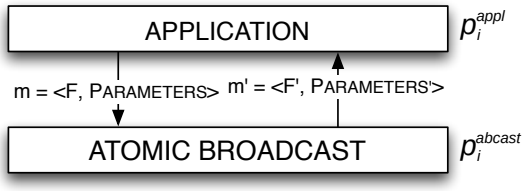


Figure 8.2: Function calls and callbacks can be modeled as messages.

p_i^{abcast}. If we represent Figure 8.1 using this message-passing model, it becomes Figure 8.3. The atomicity problem now becomes the problem to recover p_i^{appl} and p_i^{abcast} in a consistent global state.

This modeling allows us now to apply results from the existing checkpointing literature [EAWJ02]. A message m becomes *orphan* when its sender is rolled back to a state before the sending of m (m is *unsent*), but the state of its receiver still reflects the reception of m. In this case, the receiver is said to be an orphan process. Orphan processes cannot be tolerated: the orphan process needs to be rolled back (even if it did not crash). A message m is *in-transit* when its receiver is rolled back to a state before the reception of m (m is *unreceived*), while the sender is in a state in which m was sent. *In-transit* messages are tolerated under the condition that the rollback-recovery protocol is built on top of lossy channels [EAWJ02]. In our model communication is reliable, so we cannot recover in a state with in-transit messages. Therefore, in-transit messages cannot be tolerated, either.

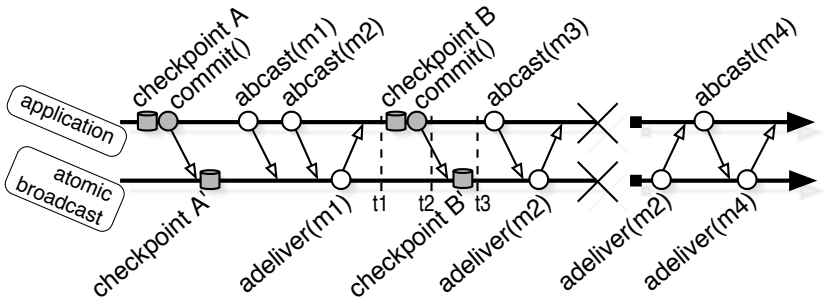


Figure 8.3: Expressing Figure 8.1 using message passing communication.

We now discuss how to recover p_i^{appl} and p_i^{abcast} in a global state with no orphan and no in-transit messages. This solution is similar to the *process-pairs* in the Tandem NonStop Kernel [Bar81]. We explain the solution in Figure 8.3. Consider the second *checkpoint-commit* pair. First of all, we can assume that no crash can occur between a checkpoint and the subsequent *sending* of the *commit*

```
1:  At application level
2:      Initialisation:
3:          nb_checks ← 0; ∀i ∈ ℕ : st[i] ← ⊥

4:      to checkpoint(state)
5:          nb_checks ← nb_checks + 1
6:          st[nb_checks] ← state
7:          st[nb_checks − 2] ← ⊥
8:          log(st); abcast.commit()

9:      upon recovery do
10:         retrieve(st); nb_checks ← max{i : st[i] ≠ ⊥}
11:         if abcast.get_nb_commits() < nb_checks then
12:             st[nb_checks] ← ⊥; log(st)
13:             nb_checks ← nb_checks − 1

14: At atomic broadcast level
15:     Initialisation:
16:         ...; nb_commits ← 0; ...

17:     procedure get_nb_commits()
18:         return(nb_commits)

19:     upon commit() do
20:         ...
21:         nb_commits ← nb_commits + 1
22:         log(nb_commits)
23:         ...

24:     upon recovery do
25:         ...; retrieve(nb_commits); ...
```

Figure 8.4: Keeping local consistency between the atomic broadcast protocol and the application.

message.[2] This reduces the problem to three cases: (1) crash at $t1$, i.e., before the checkpoints B and B', (2) crash at $t2$, i.e., after checkpoint B but before checkpoint B', and (3) crash at $t3$, after the checkpoints B and B'.

Cases (1) and (3) are analog: there are no in-transit and orphan messages in the global state (A, A') and in the global state (B, B'). Thus, we only discuss case (3). Since there is no in-transit message in the global state (B, B'), p_i^{appl} is rolled back to the checkpoint B and p_i^{abcast} is rolled back to the checkpoint B'.

In case (2), the global state (B, A') contains at least one in-transit message (the commit), therefore p_i cannot be rolled back to (B, A'). So p_i^{appl} is forced to rollback to checkpoint A and p_i^{abcast} is rolled back to checkpoint A'.

So the only problem is to know whether case (2) or (3) occurs. This can easily be done by counting the number of s_i^{appl} checkpoints and the number of s_i^{abcast} checkpoints. If the two numbers are equal we are in case (3). Otherwise, we are in case (2). Figure 8.4 shows the corresponding pseudo-code. At the atomic broadcast level, i.e., p_i^{abcast}, the variable $nb\_commits$ counts the number of commits executed so far. Its value is logged with the data that the commit procedure logs, so it really reflects the number of commits executed despite crashes. At the application level, i.e., p_i^{appl}, the array st represents the sequence of checkpoints of s_i^{appl}. The variable nb_checks keeps track of the number of checkpoints done so far. It is important that no message is adelivered during the checkpointing phase (lines 4 through 8). Upon recovery (atomic broadcast must recover before the application), st is retrieved and the value nb_checks is computed (line 10). Then, p_i^{appl} queries p_i^{abcast} to find out whether (a) it can resume execution from its very last checkpoint, or (b) it has to roll back to the previous checkpoint. Note that actually p_i^{appl} only keeps the two most recent checkpoints.

[2] Upon recovery, if we find a checkpoint, we assume that a commit was sent immediately after.

8.4 Solving Uniform and Non-Uniform Atomic Broadcast

There are several alternatives to solving atomic broadcast in the crash-recovery model, in the same way as there are various algorithms that have been proposed to solve it in the crash-stop model [DSU04]. In this section, we have chosen to illustrate how to implement the new atomic broadcast specifications of Section 8.2 by reduction to a sequence of consensus. This technique is well accepted in the crash-stop model, which justifies our choice. We first present an algorithm that implements uniform atomic broadcast in the crash-recovery model. Then, we discuss how to convert this algorithm into a more efficient one that satisfies the weaker non-uniform atomic broadcast specification. Both algorithms solve the problem by reduction to consensus.

8.4.1 Building Blocks

The algorithms we present below rely on the following building blocks.

Logging. During normal execution, processes use non-persistent memory to keep their state. They access stable storage from time to time to save data from non-persistent memory. When a process crashes and later recovers, only the data saved to stable storage is available for retrieving. A process uses function *log(*X*)* to log the content of variable X to stable storage, and the function *retrieve(*X*)* to retrieve (upon recovery) the previously logged value of X. These two functions are very costly and should be used as sparsely as possible.

Fair-Lossy Channels. Processes communicate using channels. Because of the crash-recovery model, we cannot assume reliable channels. Indeed, consider processes p and q: if p sends a message m to q while q is down, the channel cannot deliver m to q. So we assume fair-lossy channels and the two communication primitives: send($message$) *to* $destination$ and receive($message$) *from* $source$. They ensure the following property: if p sends an infinite number of messages to q and q is good, then q receives an infinite number of messages from p. Fair-lossy channels can be implemented without access to stable storage.

Consensus. The algorithms below solve atomic broadcast by reduction to consensus, i.e., we need a building block that solves consensus. In consensus, each process proposes a value, and (1) all good processes decide a value, (2) this value is the same for all processes that decide,[3] and (3) it is the initial value proposed by some process. With these properties, a process can decide several times (if it crashes and recovers) but its decision must always be the same. Section 8.4.4 discusses how to solve consensus.

[3] Actually, this defines uniform consensus. In this chapter, consensus always stands for *uniform* consensus. Note that the specification of non-uniform consensus in the crash-recovery model [ACT00] is not well-adapted for this work.

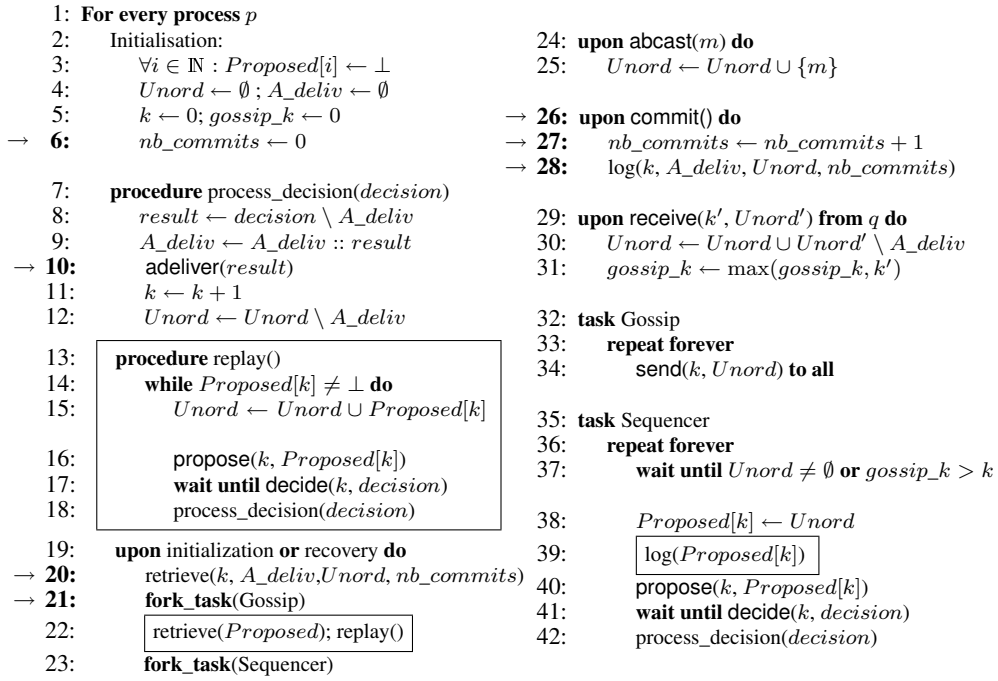


Figure 8.5: Solving uniform atomic broadcast. Small arrows mark the differences with [RR03]. Non-uniform atomic broadcast is obtained by removing the code inside the boxes.

8.4.2 Uniform Atomic Broadcast

Overview. The algorithm depicted in Fig. 8.5 implements the uniform variant of our atomic broadcast specification. The algorithm reduces atomic broadcast to a sequence of consensus as in [CT96] for the crash-stop model. It is also influenced by the algorithms in [RR03] (which are actually derived from [CT96]). The algorithm has two tasks: the *Sequencer* task and the *Gossip* task. The *Sequencer* task executes a sequence of consensus to decide on the delivery order of messages, while logging every value proposed to stable storage. For clarity, the presented algorithm proposes full messages to consensus, although it can be optimized to use only message IDs [ES06]. The *Gossip* task is responsible for disseminating new messages among all processes. This is necessary to ensure eventual message reception with fair-lossy channels. When *commit* is executed, the algorithm also logs the part of its state that is necessary in the case of a crash followed by a recovery. Upon recovery, the algorithm "replays" (see lines 13 to 18) all messages *adelivered* beyond the most recent *commit* executed before the crash. This is needed in order to satisfy the specification of uniform atomic broadcast.

Innovations. Since the basic idea of the atomic broadcast algorithm is inspired by [CT96] and [RR03], we find it inappropriate to explain the details. More inter-

esting is to focus on the differences. Thus, we now explain the main differences between the algorithm in Fig. 8.5 and the algorithm presented in [RR03] (and we also point out a bug in this algorithm, see below). We highlight these differences with small arrows to the left of the involved lines (e.g., line 6, line 10, etc.). The rectangles surrounding part of the code should be ignored for the moment (e.g., lines 13 to 18): they are discussed in Section 8.4.3.

The only new variable is $nb\_commits$ (line 6), which counts the number of *commits* locally performed since system start-up time (see Section 8.3). This variable is accessed in lines 20, 27, and 28.

Lines 26 through 28 are executed upon *commit*. Commit saves to stable storage all data necessary to restore its state upon recovery. These data are (1) the number of the current instance of consensus (or the next one, if there is no consensus running at the local process), (2) the variable A_deliv containing messages already adelivered, (3) the set $Unord$ of messages received but not yet adelivered,[4] and (4) the variable $nb\_commits$ defined above. The rest of the state is either not needed (variable $gossip\_k$), or logged elsewhere (the array $Proposed$, which stores the values proposed to consensus).

Note that, unlike [RR03], our algorithm *does* include the primitive *adeliver* (see Section 8.2). *Adeliver* occurs every time a message is added to set A_deliv, i.e., in line 10.

Upon recovery, procedure replay proposes again the initial values that were proposed before the crash. It does so in line 16. This line is necessary because we assume the consensus specification for the crash-recovery model [ACT00] (see Sect. 6.3.4), and thus, each consensus k that decided before the crash need to be restarted upon recovery to have the guarantee that consensus k decides again after the recovery. Line 16 would not be necessary if consensus specification also considered the commit primitive.

Finally, line 21 differs from [RR03]: it is incorrect in the optimized algorithm in [RR03] (if line 21 is placed after the call to *replay*, *replay* may block forever in line 17).

The correctness argument of the algorithm in Fig. 8.5 is similar to [RR03].

8.4.3 Non-Uniform Atomic Broadcast

Informally, the difference between the uniform and non-uniform atomic broadcast algorithms is that the non-uniform algorithm only needs to write to stable storage upon execution of *commit*. The uniform algorithm presented in the previous section needs to log every value proposed to consensus (Fig. 8.5, line 39). The reason is that volatile events have to be replayed upon recovery, exactly as they occurred before the crash. The uniform algorithm thus accesses the stable storage every time an instance of consensus is started. In contrast, the non-uniform algorithm can *forget* volatile events at any process, while still fulfilling its specification. Thus, if a

[4] This differs from [RR03], where this information is logged every time a new message is abcast, which is less efficient.

process crashes and recovers, it only needs to remember its state at the time of the last commit. Applications that can afford losing uncommitted parts of the execution can typically benefit from non-uniform atomic broadcast. Note that the total order and agreement properties *do hold* at good processes even if processes forget volatile events when crashing, so the application state does not become inconsistent at those good processes. The non-uniform algorithm is easily derived from Fig. 8.5 by removing the code in the white boxes (e.g., lines 13 to 18, line 22, etc.).

Access to stable storage is extremely expensive and should be used as sparsely as possible. Thus, if the application does not execute *commit* frequently, the performance of the non-uniform algorithm is highly improved compared to the uniform algorithm. Furthermore, if the underlying consensus algorithm does not access the stable storage too frequently, the performance of non-uniform atomic broadcast can even be close to a crash-stop atomic broadcast algorithm. We discuss this issue in the next section.

8.4.4 Which Consensus Algorithm Should Be Used?

Both atomic broadcast algorithms presented require an algorithm solving consensus in the crash-recovery model. Aguilera *et al* propose two such algorithms: one of them accesses stable storage, whereas the other does not [ACT00].

Consensus with Access to Stable Storage. The consensus algorithm with access to stable storage is well suited for uniform atomic broadcast. It solves consensus as long as a majority of processes are good, but accesses the stable storage very often: (1) every time the state changes locally, and (2) when the process decides. The stable storage is thus accessed at least twice per consensus. This does not impact performance of uniform atomic broadcast as much as one could think, since the uniform atomic broadcast itself logs its proposed value at the beginning of every consensus.

However, using this algorithm with our non-uniform atomic broadcast is overkill. It reintroduces frequent access to stable storage that we managed to suppress with our non-uniform algorithm, i.e., performance of the non-uniform atomic broadcast becomes poor: the performance of non-uniform atomic broadcast algorithm is almost the same as the performance of the uniform atomic broadcast algorithm.

Consensus without Access to Stable Storage. Aguilera *et al* show that consensus can also be solved in the crash-recovery model without accessing stable storage. This consensus algorithm suits our non-uniform atomic broadcast in the sense that it does not reduce performance, as it does not access stable storage. With this solution, we achieve our goal of avoiding access to stable storage as long as commit is not executed. However, the algorithm requires the number of *always-up* processes to be larger than the number of bad processes [ACT00]. This is not a big constraint from a practical point of view. Indeed, by having *commit* log part of the

state of the consensus algorithm, the *always-up* processes are only required to stay up between two consecutive *commits*.

8.5 Performance Evaluation

We have implemented different atomic broadcast algorithms to compare their performance for various group sizes: $n = 3$ and $n = 7$.[5] The algorithms implemented are: (a) the optimized uniform atomic broadcast algorithm proposed by Raynal and Rodrigues [RR03], (b) the uniform atomic broadcast algorithm of Section 8.4, (c) the non-uniform atomic broadcast algorithm of Section 8.4, and (d) a well-known uniform atomic broadcast algorithm in the crash-stop model [CT96]. All these algorithms reduce atomic broadcast to a sequence of consensus: algorithms (a) and (b) use a crash-recovery consensus algorithm that accesses stable storage (see Section 8.4.4), algorithm (c) uses a crash-recovery consensus that does not access stable storage (see Section 8.4.4), algorithm (d) uses a crash-stop consensus algorithm [CT96]. All algorithms were implemented in Java and follow Fortika's conventions (see Sect. 9.2). These conventions allow protocol composition with different composition frameworks. We used the Cactus [BHS98, HS98] framework for these experiments. Algorithms (a), (b) and (c) use the same libraries for stable storage and for fair-lossy channels. Algorithm (d) uses TCP-based reliable channels.

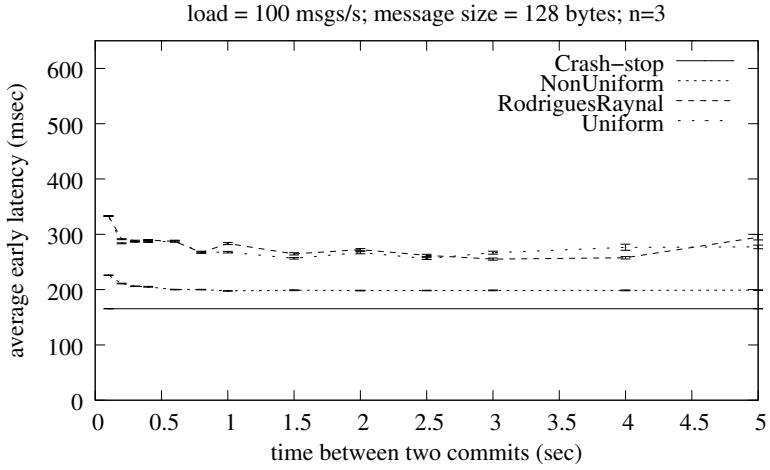
The hardware used for the measurements was (1) a 100 Base-TX Ethernet, with no third-party traffic, (2) seven PCs running Red Hat Linux 7.2 (kernel version 2.4.18-19). The PCs have a Pentium III 766Mhz processor, 128 MB of RAM, and a 40 GB (Maxtor 6L040J2) hard disk drive. The Java Virtual Machine was Sun's JDK 1.4.0.

In our experiments, the first process in the group (i.e., the process with the smallest id) steadily abcasts 128-byte-long messages.[6] The offered load was constant at 100 messages per second (i.e., the benchmark *tries to* abcast 100 messages per second, but protocol flow control will block it from time to time). The actual throughput was less than that, since the sending thread is blocked when there are too many messages in the local set $Unord$ (Fig. 8.5, line 25). Besides, all processes execute *commit* every t seconds, where t ranged from 100 milliseconds to 5 seconds. In each experiment, we measured the average *early latency* of messages after the execution became stationary. The *early latency* for message m is the time elapsed between the abcast of m and first adelivery of m [Urb03]. We also measured the average throughput, defined as the number of messages adelivered per second. Note that the experiments where performed with no crashes and no false crash suspicions. The main goal of these experiments was to see how the performance is affected as the frequency of commits increases.[7]

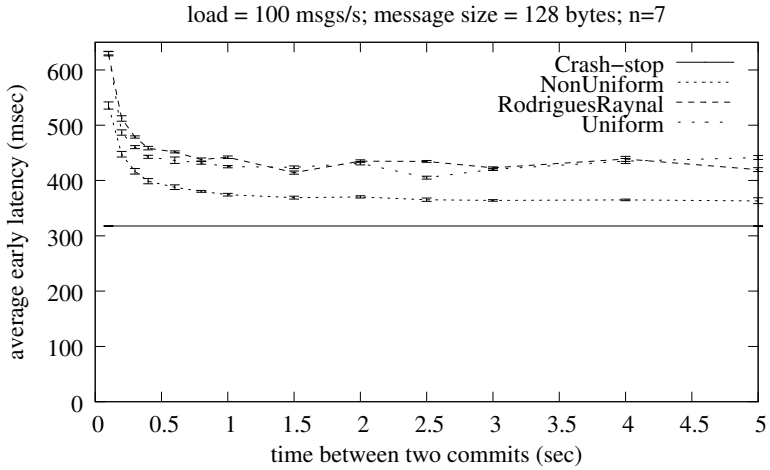
[5] We did the same tests for $n = 5$ and the results are in-between.

[6] We have also done the same experiments with multiple senders, yielding similar results.

[7] For Rodrigues-Raynal [RR03], a checkpoint is taken instead of a commit, which is the same in



(a) group size: 3 processes

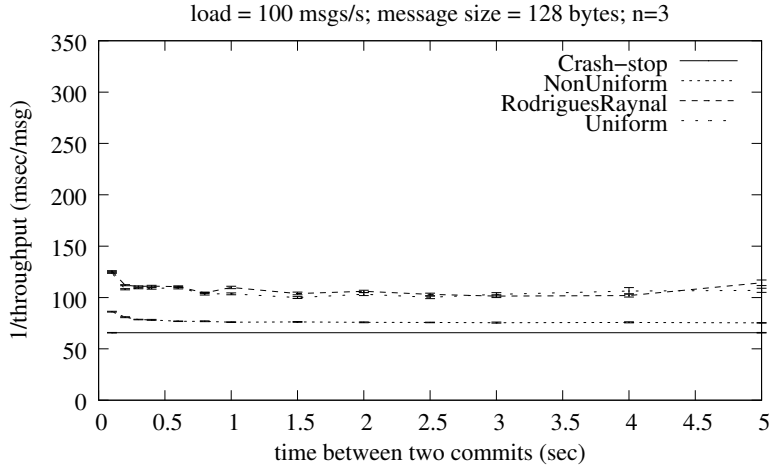


(b) group size: 7 processes

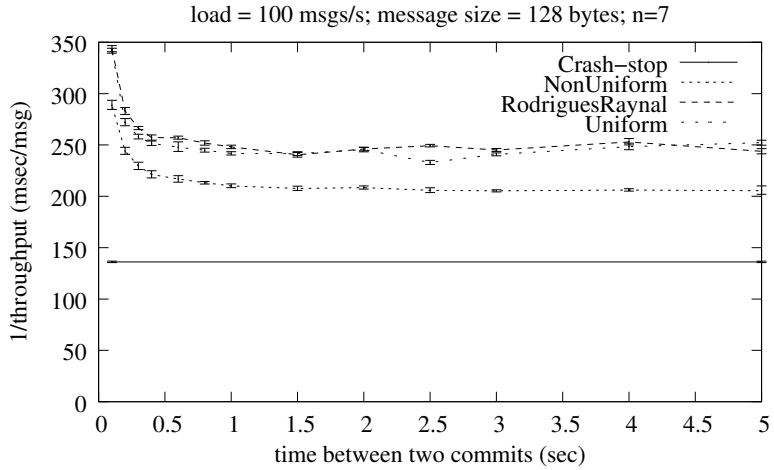
Figure 8.6: Early latency of various atomic broadcast algorithms

Figure 8.6 shows the early latency results with the 95% confidence interval. As expected, Rodrigues-Raynal and our uniform algorithm perform similarly since both of them use the stable storage for every consensus. An important observation is that the non-uniform algorithm performs much better than the two uniform algorithms when commits are not frequent, since it only accesses the stable storage when executing commit. The performance of the non-uniform algorithm can even compete with the crash-stop atomic broadcast algorithm (which does not access stable storage at all). As the commit period reduces, the performance of all crash-recovery algorithms, including the non-uniform algorithm, degrades asymp-

terms of implementation.



(a) group size: 3 processes



(b) group size: 7 processes

Figure 8.7: 1 / throughput of various atomic broadcast algorithms

totically, since access to stable storage becomes more and more frequent.

The throughput results are shown in Figure 8.7, also with the 95% confidence interval. Actually, in order to compare the curves of Figures 8.6 and 8.7, we have plotted the values of *1 / throughput* in Figure 8.7. We can observe that the results in the two figures are very similar. Note that the performance obtained is not spectacular: our proof-of-concept implementations have not been optimized, e.g., they all make extensive use of the standard Java serialization, which is known to be very inefficient (see Sect. 3.3.3). However, this does not preclude the performance comparison of the different algorithms.

Figures 8.6 and 8.7 do not show directly the impact of *commit* on the latency.

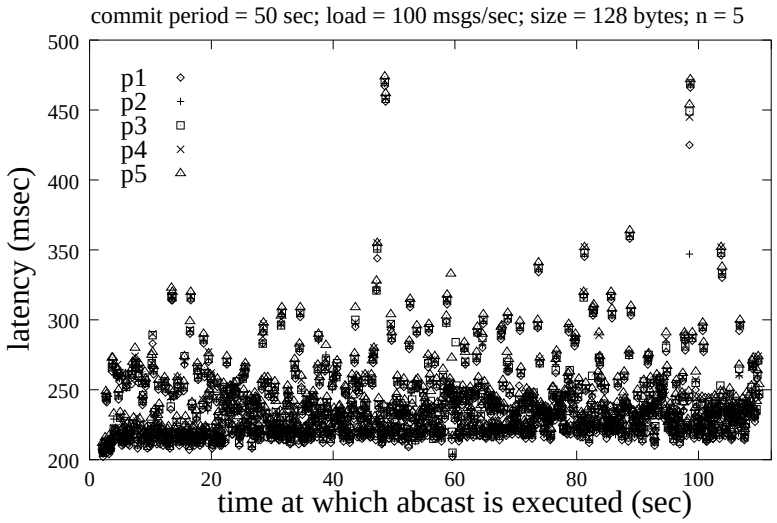


Figure 8.8: Latency in a single experiment with the non-uniform atomic broadcast algorithm.

This can be seen in Figure 8.8. The figure corresponds to one single experiment with the non-uniform atomic broadcast algorithm for a group of size $n = 5$. The figure shows the latency, once the steady state is reached, as a function of the time at which the abcast is issued. The figure shows *all* latencies, not only the early latency: if the $abcast(m)$ is issued at time t and m is adelivered at p_1 at time $t + \Delta_1$, at p_2 at time $t + \Delta_2$ at p_2, etc., we plot five dots with coordinates $(t, \Delta_1), (t, \Delta_2), \ldots, (t, \Delta_5)$. In Figure 8.8, *commit* was executed approximately at $t = 50$ and $t = 100$. We can clearly observe how latency is affected by the execution of *commit*. The high latencies around $t = 50$ and $t = 100$ come from messages that were already abcast but not yet adelivered when the commit operation started: the latency of these messages was affected by the commit operation.

8.6 Conclusion

We have proposed two novel specifications of atomic broadcast in the crash-recovery model, for uniform and non-uniform atomic broadcast. The key point in these two specifications is the distinction between permanent and volatile events. This distinction allows us to properly define the concept of non-uniformity in the crash-recovery model. Despite some attempts in the literature [ACT00, BG00], the concept of non-uniformity in the crash-recovery model did not have so far a satisfactory definition. We have also pointed out the problem of process recovery after a crash, where the application state needs to be consistent with the state of the atomic broadcast algorithm. We have shown how this problem can be solved. It is important to understand that this consistency problem does not arise in the

crash-stop model, which explains that it was overlooked up to now. Finally, we have run experiments to compare the performance of the two new atomic broadcast algorithms with two other algorithms, one based on the crash-stop model, the other based on the crash-recovery model.

Part III

Putting It All Together: *Fortika*

Chapter 9

The Fortika Group Communication Toolkit

This chapter presents Fortika, a group communication toolkit written in Java. It is the main prototype implementation of the thesis. All architectural and algorithmic contributions presented in the previous chapters have a proof-of-concept implementation in Fortika. Its algorithmic code is not implemented for a particular protocol composition framework, which in theory permits the use of any event-driven protocol composition framework. To this day, Fortika includes compositions providing atomic broadcast for three different system models.

9.1 Introduction

The key idea behind Fortika's design is to isolate the algorithmic code from the code related to protocol composition. Thus, Fortika does not define its own protocol composition framework, but rather a set of conventions that protocol programmers must follow. According to these conventions, every protocol: (1) is a Java class containing the algorithmic code, (2) consists of a set of handlers and private state, and (3) reacts to events exclusively. The third-party framework, (1) routes events from one protocol module to the following one, (2) provides special services, like flow control, timers, etc, and (3) is responsible for interacting with the environment (the application and the network).

Three complete compositions providing atomic broadcast have been implemented in Fortika, one for each of the following system models: static crash-stop, dynamic crash-stop, and static crash-recovery. The application interface is easy to use: it consists of three methods for joining/removing processes (in dynamic compositions) and abroadcasting messages. Additionally, the application implements a set of callback methods for view changes (in dynamic compositions) and message adelivery.

To the best of our knowledge, Fortika is the first group communication toolkit to offer atomic broadcast semantics in the crash-recovery model.

9.2 Conventions for Obtaining Framework-Independent Code

Almost all existing modular group communication toolkits that use a general-purpose protocol composition framework are optimized for that framework: the protocol code has been written bearing in mind that it will be composed using that particular framework. As a result, the part of the code that is strongly dependent on the framework (triggering events, binding events to handlers, etc.) and the purely algorithmic code are usually mixed up. This has an inherent drawback: protocol modules are barely reusable in other protocol composition frameworks. The essential idea behind Fortika's design is to increase reusability of protocol modules across frameworks to a maximum extent.

The Fortika prototype implementation does not include any protocol composition framework, it rather uses already existing ones. The basic requirement for composing Fortika protocol modules with a third-party framework, is that the latter must use events for interactions between protocol modules (see Sect. 3.3.1.1). Moreover, protocol programmers have to follow a set of strict conventions when writing the protocol if they want their protocol modules to be composable in the context of Fortika. The aim of these conventions is making the algorithmic code independent from the code related to the protocol composition framework.

The Protocol Code. Every protocol module is implemented as a Java class. The protocol state corresponds to private attributes of that Java class. Event handlers are public methods, and the method bodies contain the algorithmic code.

A set of protocol modules implemented in this way can not be put together directly. Every class implementing a protocol module is wrapped by a framework-dependent Java class that represents the protocol module in the framework chosen (e.g., class Microprotocol in Cactus, classes Layer and Session in Appia, etc.). The wrapper class is usually a very simple class with few lines of code. It is responsible for routing incoming events to the appropriate handler, routing outgoing events to the framework, and any other task required by the framework for correct composition .

Incoming events are usually handled in the following manner. The wrapper class defines a set of event handlers (using the framework's facilities) and binds them to the appropriate event types. The code within these handlers is simplest: a method call to the handler in the algorithmic class. Outgoing events, i.e, events triggered by the protocol module are managed as follows. The constructor of the algorithmic class accepts a callback method to be called when the algorithm needs to trigger an event. This callback method contains two parameters: the event type and its arguments. This callback is typically implemented by the wrapper class: it creates and triggers an event with the type and arguments using framework-dependent mechanisms. The transitions from the wrapper to the algorithmic class and vice-versa are implemented using method calls, so they are very cheap.

Concurrency Model. In Chapter 4 we argued that the concurrency aspect of a composition cannot be solved at the level of protocol modules (i.e., by the protocol programmers on their own). In Fortika, protocol programmers write their code without worrying about concurrency: it is up to the framework to solve race conditions, deadlocks, etc. Hence, the solution implemented requires the protocol composer's intervention.

In particular, the code of every protocol module assumes that only one event is being handled at a time within the whole composition. Moreover, if two events e_1 and e_2 are triggered in the same handler, it assumes that any event e'_1 that causally depends on e_1 is executed before any event e'_2 that causally depends on e_2. In short, protocol programmers assume that *extended causal order* is enforced (see Sect. 4.6.2) when writing their code.

To enforce these properties at the level of the composition, the approach depends on the framework being used:

- *Appia*. By default, Appia allows no concurrency inside the composition. It enforces an order of events very close to extended causal order. Besides, the framework design ensures that event execution never overlaps. Therefore, little needs to be done when composing Fortika protocols.

- *Cactus*. The approach for Cactus is the following. The whole composition behaves like a monitor: all threads that execute code inside the composition do so in mutual exclusion. The protocol code can only *invoke* events, and never *raise* them (see Sect. 2.3.2). The advantage of event invocation is that the same thread executing the triggering protocol module executes a method call and thereby jumps to the handling protocol module. The use of method calls for event triggering has an additional advantage: it provides extended causal order for free. However, to avoid overlapping execution of handlers, which is an undesirable behavior, Fortika protocol modules must follow the conventions presented in Sect. 4.5.

- *Samoa*. In Sections 2.3.3 and 4.3, we have described Samoa's transparent concurrency model. Protocol modules have the impression that there is no concurrency in the composition, hence, nothing is needed to support Samoa. Method calls are used for event triggering, which enforces extended causal order, but requires the same programming conventions as in Cactus (so that execution of handlers does not overlap).

In all three cases, protocol modules must be *reactive* (see Sect. 4.3): (1) they should not spawn new threads, (2) their handlers should not block for a long time (otherwise, the whole composition may block), and (3) direct interaction with the outside world (which may block on I/O operations) is not permitted.

9.3 Compositions Implemented

All algorithms and architectures presented in Part II have been implemented in Fortika. Currently, there are three different compositions for each of the three system models: the crash-stop model with static and dynamic groups, and the crash-recovery model with static groups.

9.3.1 Static Crash-Stop Model

This composition was the first to be implemented, and allowed us to compare Appia and Cactus[1] (see Sect. 3.2). The protocol modules of this composition have been described in Chapter 3.

9.3.2 Dynamic Crash-Stop Model

Chapter 7 described a new architecture for group communication in the dynamic crash-stop model. The architecture was implemented in Fortika, yielding the composition shown in Fig. 9.1. The protocol modules marked with a gray background in the figure were reused from the basic static composition (see Sect. 9.3.1). These modules implement protocols whose operation remains the same when switching from static to dynamic groups. The remaining modules are briefly described below:

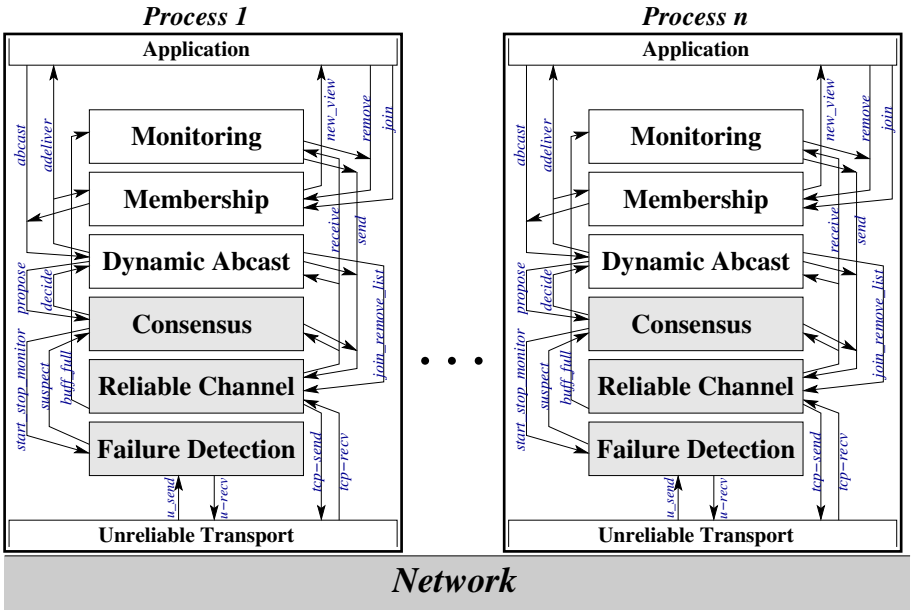


Figure 9.1: Atomic Broadcast. Composition in Fortika for the dynamic crash-stop Model.

[1] A preliminary comparison of Appia and Cactus was performed in [Men01, WMS02a].

- *Monitoring*: the module is responsible for converting a process suspicion into a process exclusion. In that case, this protocol module triggers a *remove* event on the Membership module.

- *Dynamic Atomic Broadcast*: implements uniform dynamic atomic broadcast [Sch06]. Upon view change, it provides the *same view delivery* property (see Sect. 6.3.6). It relies on the Consensus protocol module to achieve total order of the delivered messages and view changes, and on the Reliable Channel protocol module to disseminate messages.

- *Membership*: enables processes to join/leave the group. It implements primary-partition membership (see Sect 6.2.3). The module depends on the Dynamic Atomic Broadcast and Reliable Channel protocol modules.

The most noteworthy feature of this composition coincides with the architectural contribution presented in Chapter 7: atomic broadcast should be solved first, and then group membership. Thus, (dynamic) atomic broadcast becomes aware of changes to the group of processes and provides total order of view changes along with total order of application messages. This provides several advantages (see also Sect. 7.4):

- The composition is less complex because all ordering problems are solved at the same level (atomic broadcast): (1) ordering application messages, (2) ordering view changes, and (3) ordering messages with respect to view changes (i.e., same view delivery property).

- In traditional compositions, atomic broadcast always depends on group membership, that is, membership is below atomic broadcast. In that case, the atomic broadcast algorithm is not fault-tolerant itself, and gets blocked in the presence of process crashes until group membership excludes faulty processes. In the new composition, atomic broadcast relies on a fault-tolerant consensus algorithm, which makes it fault tolerant as well. As a result, atomic broadcast is able to continue operation even in the presence of crashes. This makes the whole composition more responsive.

As discussed in Chapter 7, the stack can be extended with a Generic Broadcast protocol module. This is considered as future work (see Sect. 11.2).

9.3.3 Static Crash-Recovery Model

The third composition implemented in Fortika is depicted in Fig. 9.2. It provides crash-recovery atomic broadcast to the application. It assumes static groups: the group is configured at system start-up time and cannot be changed afterwards. Hence, there is no membership protocol module. To the best of our knowledge, Fortika is the only group communication toolkit to offer atomic broadcast in the crash-recovery model.

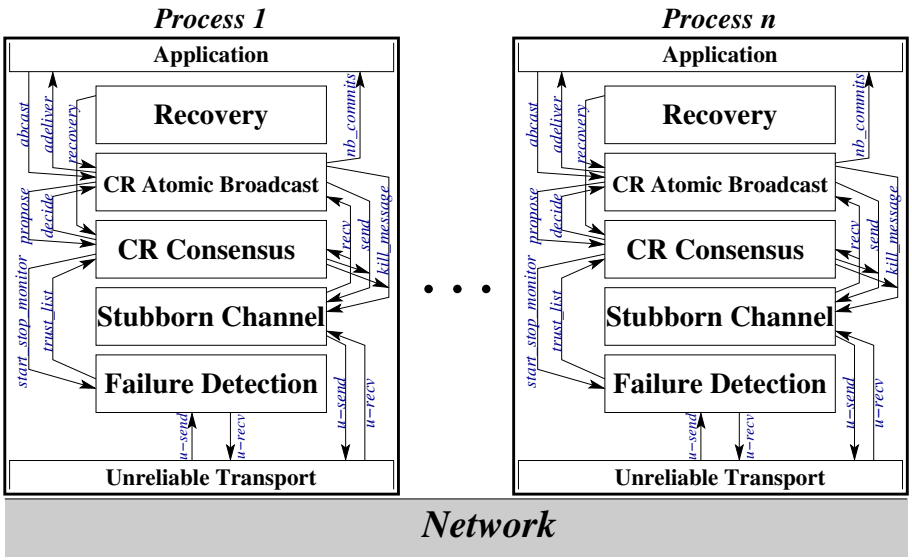


Figure 9.2: Atomic Broadcast. Composition in Fortika for the Static/crash-recovery Model.

With respect to the two previous compositions, this one represents a major model shift. From an algorithmic point of view, the fact that processes are able to recover implies that all protocols need to be redesigned. This is why this composition could not reuse any protocol module from the previous compositions. Here is a brief description of the protocol modules that are part of this composition:

- *Recovery*: detects (by accessing the disk) whether the process is started for the first time or is recovering from a crash.
- *Atomic Broadcast*: implements the atomic broadcast specification for the crash-recovery model (see Sect. 8.2). The composer can choose between three versions: the uniform and non-uniform versions presented in Chapter 8, and the algorithm of [RR03]. All implementations use the Stubborn Channel and the Consensus protocol modules.
- *Consensus*: implements consensus for the crash recovery model [ACT00]. It uses the Stubborn Channel protocol module.
- *Stubborn Channel*: yields point-to-point communication with the *No-creation* and *Stubborn* properties defined in Sect. 6.3.1. It uses a low-level protocol module (not shown in Fig. 9.2) that provides fair-lossy channels.
- *Failure Detection*: provides failure detector $\diamond\mathcal{S}_u$, proposed for the crash-recovery model [ACT00].

The main advantages of crash-recovery protocols are the following (see Chapter 8):

- The application can checkpoint its (replicated) state in synchrony with the composition. This makes such replicated state less volatile. Assume, for an illustration, that all replicas crash at a given time. In the crash-stop model, as the replicated state is stored in main memory (checkpoints synchronized with the composition are not supported), it would be gone forever. In the crash-recovery model, the checkpoints of the application state (performed together with the execution of the *commit* primitive) would permit crashed processes to continue operation after recovery.

- Even if not all processes crash, the advantage of crash-recovery protocols is evident. In the (static) crash-stop model, a majority of correct processes must be correct so that consensus (and atomic broadcast[2]) can make progress. In the (static) crash-recovery model, still a majority is needed, but a majority of *good* processes [ACT00]. As presented in Sect. 6.2.4.2, good processes can crash, as long as they recover after some time. As a result, if a majority of processes is down at a given moment, the execution gets blocked, but some process recoveries are enough so that the composition can make progress again [ACT00].

As there is no free lunch, these advantages come at the expense of frequent log operations to disk, which is expensive in terms of performance. This makes crash-recovery protocols much slower than their crash-stop equivalents. However, as proposed in Chapter 8, if the application can afford non-uniformity we can get the best of two worlds: a non-uniform atomic broadcast algorithm that accesses the disk sparsely, which yields good performance, close to that of crash-stop protocols.

For the moment, the composition is designed for the static model, yet it is possible to extend it to a (crash-recovery) dynamic model in the same way as we extended composition of Sect. 9.3.1 to that of Sect. 9.3.2.

9.4 Relevant Implementation Issues

In this section, we discuss some issues related to the implementation of Fortika.

9.4.1 Interface with the Application

Relevant group communication toolkits, such as Ensemble or JavaGroups, offer a powerful interface to the user, allowing him to specify the sequence of protocol modules (or a set with their properties) that compose the stack. When experimenting with such interface, our feeling was that the user has to be familiar enough with the implementation of the protocol modules if she is to be sure that the stack

[2]Consensus and atomic broadcast are equivalent problems [CT96].

is correct. In some occasions, we even ended up using some by-default composition appearing in a tutorial because of the lack of success when trying to customize a stack.

When we were confronted to the design of Fortika's interface with the application, we wanted it to be as simple to use as possible. The key issue here is that what we call *composer* in this thesis (the person that puts protocol modules together to build a composition) is not necessarily the same actor as the *user*, which maybe only wants some predefined composition that works. As a result, every different composition supported by Fortika is represented by a different Java class that acts as a proxy between the application and the composition. For each composition, the primitives available to the application can change (for instance, static compositions do not define primitives join or remove). We have exploited Java class inheritance to minimize repetition of primitives in the interface.

From the point of view of the user, every composition in Fortika is represented by two classes: the proxy class, and the callback class. The former contains the primitives called from the application to the composition, whereas the latter contains callbacks that the composition uses to communicate with the application.

We now present an example. The following line

abcast = **new** DynamicAbcast(v_0, callbacks)

creates a (local) composition offering atomic broadcast in the dynamic model (see Sect. 9.3.2), with view v_0 as initial view:

The application uses object abcast to communicate with Fortika:

- abcast.send(m) to abroadcast message m.
- abcast.join(q) to request process q to be joined to the group.
- abcast.remove(q) to request process q to be removed from the group.

Besides, the application provides object callbacks, which implements interface DynAbcastCallbacks and contains the following methods:

- adeliver(m, q) notifies of the adelivery of message m whose sender was q.
- new_view(id, v_{id}) notifies that view number id has been installed.

9.4.2 Interface with the Network

In Fortika, as in any group communication toolkit, low-level protocol modules need to access the network. The problem arises when a reactive protocol module[3] needs to use TCP sockets, with blocking methods like send or recv.

To solve this problem, Fortika includes a library for non-blocking access to TCP sockets. This library contains only non-blocking methods, and allows the

[3]For a definition of *reactive protocol module*, see Sect. 4.3.

management of external threads from inside the composition. So, using only non-blocking methods, a protocol module can: (1) create server or client sockets, (2) connect to remote server sockets, (3) handle closed or broken connections, (4) launch external threads that send/receive messages or that accept incoming client connections, and (5) stop those threads. For further details on this library we refer the reader to Fortika's Javadoc files.

9.4.3 Flow Control

When the application issues requests so fast that the composition can not keep up with it, a flow control mechanism is needed, otherwise the system is not stable. A good flow control mechanism should not be activated unless the composition is operating near the system limits. We believe that protocol modules do know best when the composition can not keep up with the workload generated by the application. For instance, if an atomic broadcast protocol realizes that more messages are being abcast than adelivered, it is seemingly a situation where the application must be slowed down.

Fortika provides flow control to the protocol modules via a very simple library. Some protocols are not related at all to flow control (e.g., failure detection). A protocol module interested in using flow control registers for it using library method getFreshKey, which returns an integer key used by the library to identify that module. The flow control can be in one of the following two states: *open* or *blocked*. A registered protocol module executes method block whenever it requires flow control to be in a blocked state, and method release when this requirement disappears. As there can be several protocol modules using it, the flow control is in blocked state if there is at least one protocol module that demands it, otherwise it is open. When the application accesses the composition code (via one of the methods described in Sect. 9.4.1), method enter is called. If flow control is open this method has no effect, but it will suspend the calling thread if the flow control is blocked.

In summary, the flow control library gives protocol modules the opportunity to block the application temporarily when it can not keep up with the workload generated.

9.5 Conclusion

In this chapter, we have presented Fortika, a prototype group communication toolkit that played the role of testbed for the ideas developed in this thesis. Thanks to the conventions for writing framework-independent protocol code, Fortika protocol modules contain highly reusable algorithmic code, which can be used by any event-driven protocol composition framework. Fortika's modular design allows the composer to choose both at the level of individual protocols and complete compositions. There are at present compositions for three different system models, as well as several well-known algorithms for some key protocol modules: Chandra-

Toueg [CT96] or Mostefaoui-Raynal [AR01] consensus, ping-based or heartbeat-based failure detector, etc. In short, Fortika represents a modular, framework-independent solution to group communication.

There are some projects in which Fortika is used as a tool. Arnas Kupšys uses Fortika's crash-recovery atomic broadcast composition to replicate the JMSGroups server [KPSW04, KE05]. Samoa's designers use all three Fortika stacks to conduct research on transparent concurrency and dynamic protocol update [WRS04, RWS06]. Finally, Fortika is also used for research on fault injection, which is presented in next chapter.

Chapter 10

Fault Injection: Assessing the Crash-Failure Assumption of Fortika

10.1 Introduction

Group communication toolkits often assume that processes only fail by crashing (see Sect. 6.2.2), i.e., they suddenly stop their operation. This crash-failure assumption provides a powerful means to simplify algorithm design. When making this assumption, however, one should not forget that the errors that affect real-life implementations often involve complex error-propagation patterns that cannot be directly modeled with abstract crashes. Importantly, the crashes that are modeled in theory are quite different from the common notion of crashes (e.g., process termination due to a segmentation violation). The crash failure assumption was originally formulated with the caveat that systems should embed robust self-checking mechanisms (e.g., internal assertions) to provide error detection with high coverage and negligible latency. On detecting an internal error (e.g., an out-of-range value in a process variable due to a software bug or a hardware transient), the process terminates itself, and thus, converts the error into a clean crash failure. Ignoring such a discrepancy between reality and theory can lead to complex replicated applications that pay a large price in terms of performance overhead yet fail to deliver the promised dependability level.

In [BWKI03], the authors present a systematic, experimental study of the Ensemble toolkit to analyze how the system responds to a variety of real errors (e.g., bit flips in memory segments and network messages). The key result of that study is that up to 5–6% of the manifested errors involve safety/liveness property violations and error propagations to multiple processes (basically cases occurring because the crash failure assumption does not always hold). Detailed analysis of the Ensemble system shows that these failures are due to internal self-checking mechanisms with poor coverage (less than 10%). The results of that analysis are not reported as an

indictment of Ensemble, which is a well-engineered product, but rather because they probably hold for many other fault-tolerant systems.

We believe that experimental evaluation and theoretical development must go hand in hand when designing a fault-tolerant system. The present chapter leverages and extends [BWKI03] by demonstrating a *error-injection-driven design methodology* to build robust fault-tolerant systems, where error injection is integrated in the design process. The methodology is demonstrated on Fortika and operates as follows: (1) perform error injection on an early prototype of the system to identify and quantify reliability bottlenecks in the systems; (2) based on the obtained insights, enhance the system's design and implementation to reduce (ideally remove) the identified bottlenecks; (3) perform a followup error injection campaign to measure the reliability improvement obtained with the enhanced version of the system, and reiterate if necessary.

Often the designer must make important decisions (e.g., in the early stages of a design) that may have a significant impact on the overall reliability of the system yet such an impact is hard to predict beforehand. For instance, Ensemble's architects decided to write the system in the OCAML dialect of the ML language so that the code would be amenable to automated proof checking (as demonstrated in [KHH98]). In theory, OCAML may be preferable, as implementation language, to Java. In actuality, the experimental study presented in this chapter reveals the opposite. In [BWKI03] the authors show that the code formally proved for correctness (i.e., the code of the group communication protocols) can be as small as 5% of the code executed at runtime. A large number of problems originate in portions of the code on which the designer has little control (e.g., OCAML runtime support in case of Ensemble). In this study we show that the Java runtime support offers more efficient self-checking capabilities than the OCAML runtime support, which makes a system built in Java (Fortika, in our case study) significantly more reliable (less number of fail-silence violations) than a system built in OCAML. This result corroborates the validity and the importance of the proposed design methodology.

10.2 Experimental Setup

Out of the three compositions implemented in Fortika, this study focuses on the one offering atomic broadcast in the dynamic crash-stop model (see Sect. 9.3.2). We recall that such composition enforces *uniform* properties (see Sect. 6.3.3). While uniformity has been extensively studied in the literature (e.g., most group communication protocols designed under the crash failure assumption provide uniformity), the actual cost of achieving uniformity in real systems subjected to failures is not well understood. One of the contributions of this work is an experimental assessment of the difficulty implementing uniformity in realistic operational conditions.

In this study, the protocol composition framework used to compose Fortika's protocol modules is Cactus (see Sect. 2.3.2), which provides good performance as compared to other frameworks (see Sect. 3.3.3).

The experimental testbed consists of three nodes interconnected by an Ethernet 100 Mbps LAN. The nodes are Pentium III 500 MHz-based machines running Linux 2.4 and Sun's Java Virtual Machine (JDK 1.4.2). NFTAPE [SFKI00], a software framework for conducting automated fault/error injection experiments, is used to launch the experiments, monitor the test execution, and gather the output results in a log file. To create system activity during the error injection campaigns, we use as workload a synthetic *benchmark* application. The application employs Fortika's atomic broadcast to send 200 messages at a constant rate of 10 messages per second. Each application message has a fixed size of 1000 bytes. The workload executes on three machines, forming a group of three processes.

Clean Crash Injection. An initial set of 900 experiments is performed to study the behavior of Fortika applications in the presence of clean crash failures caused by sending a term, kill or segv UNIX signal, thereby forcing the target process to terminate immediately. Since Fortika is designed to cope with crash failures this type of injections is merely intended to reveal defects in the design or the implementation of Fortika's protocols. The obtained results did not reveal any flaw in the design and implementation of Fortika's crash-resilient group communication protocols.

Roadmap. Clean crash failures provide a coarse approximation of the effects that errors can have in real distributed systems. A thorough study of these effects is the objective of the following sections, where two major sets of error injection campaigns are conducted: (1) *memory injections*, to assess the impact of errors in the memory segments of a Fortika process; and (2) *network injections*, to analyze the impact of errors in the contents of messages exchanged in support of Fortika's group communication protocols. Each experimental error injection campaign is performed in three macro-steps: (i) initial set of injections to identify reliability bottlenecks of Fortika's design/implementation, (ii) removal of the identified design/implementation flaws, and (iii) final set of injections to assess the error resilience of the enhanced design/implementation.

10.3 Error Injection into Memory

The experimental setup described in Sect. 10.2 is used to study the impact of errors in a Fortika process. In each experiment we execute three copies of the benchmark application discussed in Sect. 10.2 on three different machines, and inject a single error, at a random time, in a selected process. The experiment ends when all application processes terminate and, subsequently, the system is reset.

In the experiments considered in this section, errors are injected in the process with the lowest id, namely the *coordinator* process. The coordinator process has a special role in the consensus protocol module. Upon coordinator failure, any other process p blocks until p suspects the coordinator and selects a new coordinator:

Table 10.1: Error models.

Error Model	Description
JVM TEXT	A single bit is flipped in the text segment of the JVM process.
JVM HEAP	A bit is periodically flipped (every 5 seconds) in allocated regions of the heap memory of the JVM, until the process terminates or crashes. Note that more than one error may be injected during a single experiment.
BYTECODE STACK	A single bit is flipped in the execution stack of a Java thread running in the target process.

Fortika's consensus algorithm uses the rotating coordinator paradigm to recover from a coordinator failure [CT96]. This mechanism can lead to all processes in the group to block indefinitely, as discussed later in the section. We expect injections in the coordinator to provide worst case scenarios and, in particular, to generate error propagations.

10.3.1 Error Models and Outcome Categories

Two main types of error injections are conducted:

Java Virtual Machine Errors. Fortika's Java code is compiled into Java bytecode and executed/interpreted at runtime by the Java Virtual Machine (JVM). The JVM runs as a native UNIX process and, thus, its execution can be affected by accidental errors, e.g., software bugs or hardware transients. We study the error behavior of the JVM by injecting bit errors in text and heap memory segments of the JVM of the coordinator process.

Bytecode Errors. The Java bytecode of Fortika's protocols that is executed/interpreted by the JVM is placed in a read/write memory region and can thereby be affected by random errors in memory. We study the error behavior of Fortika's bytecode by injecting single-bit errors in the stack segment of a selected Java thread running in the coordinator process.

The error models introduced above are further discussed in Table 10.1 and represent a combination of those used in several past experimental studies [Fuc98, MJ94]. By injecting single bits in the targeted process, we emulate errors in the main memory, the cache, the processor execution buffer, and the processor execution core, as well as errors occurring during transmission over a bus. Previous research on microprocessors [ROT94] has shown that most (90–99%) device-level transients can be modeled as logic-level, single-bit errors. Data on operational errors also shows that a large number of errors in the field are single-bit errors.

Manifested errors are divided in two major outcome categories: (1) *crash failures*, in which the injected process stops executing and no incorrect state transition is performed before the failure, and (2) *fail silence violations*, in which the injected process performs incorrect state transitions—this failure type covers cases such as corrupted data delivered to the application or a corrupted message sent to other processes. The two categories and their corresponding subcategories are reported in Table 10.2.

Table 10.2: Outcome categories.

Activated	The corrupted instruction/datum is executed/used.
Manifested	The corrupted instruction/datum is executed/used and does cause a visible abnormal impact on the system.
Crash Failure	*SIGNAL*: the operating system terminates the target process by sending a signal (e.g., SIGSEGV, SIGILL, SIGBUS, SIGFPE). *ASSERT*: the target process shuts itself down owing to an internal check violation detected (in the JVM code or in Fortika code). *HANG*: the target process does not terminate and does not make progress.
Fail Silence Violation	The target process misbehaves and possibly sends corrupted messages to other processes, causing them to fail.

10.3.2 Software-Based Error Injectors

Three software-based error injectors were used in the memory error injection campaign: (1) a *JVM text* injector, which utilizes the hardware debug registers available in the Pentium processors to automatically inject errors in the memory segments of a process executing the Java Virtual Machine (JVM); (2) a *JVM heap* injector, which overrides the libc shared library to intercept calls to malloc/free memory allocation routines, so that it can keep track of the allocated heap memory regions and perform periodic injections into these regions; (3) a *Bytecode stack* injector, which utilizes the debug interface provided by the JVM to automatically inject errors in the Java bytecode executed/interpreted by the JVM. The injection mechanism employed by the JVM text injector and the Bytecode stack injector includes the following steps: (1) set an instruction breakpoint to be used as a trigger address for the error injection, (2) inject an error at a target address when the breakpoint is hit a predetermined number of times, (3) monitor error activation, and (4) collect and store the outcomes from the injection for off-line analysis. The choices of the trigger breakpoint and the target address aim at maximizing the rate of error activations and are based on profiling information (discussed in the next section).

10.3.3 Profiling

To maximize the rate of error activations we profile the system execution both at the level of the Java Virtual Machine, which executes/interpretes Fortika bytecode, and at the level of Fortika bytecode, which constitutes the actual implementation of Fortika's group communication protocols.

Java Virtual Machine Profiling. Most JVM functionalities are contained in a set of shared libraries that are linked on-demand by a small front-end executable (about 64KB), which is the java program. In general, a Fortika application does not use all JVM shared libraries with the same frequency. By profiling these libraries under the workload generated by the benchmark described in Sect. 10.2 we can identify the most frequently used library functions and determine the relative importance of the different subsystems of the JVM. Profiling is done by using a

ptrace-based tool.[1] As Figure 10.1 indicates, when running a Fortika application the java executable loads four JVM shared libraries, but only three are intensively used: libjava.so, libjvm.so, and libnet.so. These three libraries contain 41,032 functions. Profiling enables us to identify the 4,196 functions that cover 99.9% of runtime function invocations to the JVM code. Trigger addresses and target addresses for JVM text injections are randomly selected from this smaller set of functions.

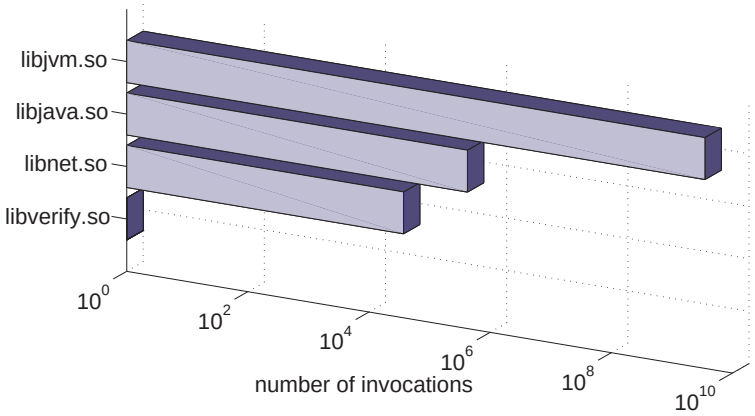


Figure 10.1: Profiling at the JVM level.

Bytecode Profiling. Figure 10.2 shows profiling information for Fortika bytecode. We observed that most of the function invocations are to the java.io library (74%) and, more specifically, to the object serialization subsystem of this library. This result is consistent with Section 3.3.3, which shows that Fortika spends a significant fraction of time in marshaling and unmarshaling network messages. We used the profiling information to select trigger addresses and target addresses for the bytecode injections.

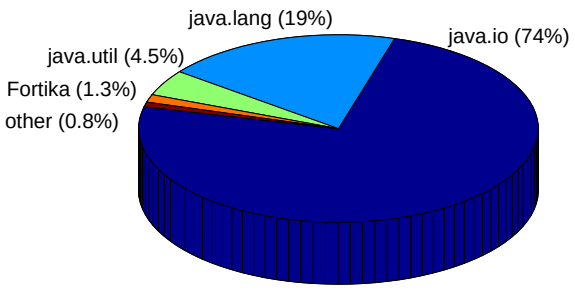


Figure 10.2: Profiling at the bytecode level.

[1] Ptrace is a process-tracing facility offered by UNIX operating systems.

10.3.4 Memory Injection Results

Table 10.3 reports the results from error injection experiments for the JVM-error model and the bytecode-error model listed in Table 10.1. Before discussing the results in detail, we point the reader to two important columns in Table 10.3: the *ASSERT* column, which corresponds to cases in which self-checking mechanisms embedded in the system are able to enforce the assumed crash failure semantics (by detecting an internal error and converting it into a self-termination); and the *Fail Silence Violations* column, which corresponds to cases in which Fortika's fault-tolerant protocols fail to tolerate the failures originating from the injected errors. We expect a robust system to maximize the number of ASSERT cases and minimize the number of Fail Silence Violation cases.

Table 10.3: Memory injection results.

Error Model	Total Injected Errors	Total* Activated Errors	Total† Manifested Errors	Manifested Errors‡			
				Crash Failures			Fail-silence violations
				SIGNAL	ASSERT	HANG	
JVM Heap	14490	N.A.	2138	1783 (83%)	320 (15%)	34 (1.6%)	1
JVM Text							
libjava.so	2055	1491 (73%)	1134 (76%)	811 (72%)	200 (18%)	59 (5.2%)	64 (6%)
libjvm.so	2600	711 (27%)	531 (75%)	406 (76%)	88 (17%)	20 (3.8%)	17 (3%)
libnet.so	2217	719 (32%)	536 (75%)	363 (68%)	115 (22%)	28 (5.2%)	30 (6%)
Bytecode Stack	5466	3294 (60%)	2009 (61%)	34 (1.7%)	1428 (71%)	21 (1.0%)	526 (26%)

* The ratio activated/injected is shown in parenthesis.
† The ratio manifested/activated is shown in parenthesis.
‡ Percentages with respect to total manifestations are shown in parentheses.

JVM Error Injections. The heap-injection results of Table 10.3 show a relatively low manifested/injected ratio (2138/14490=0.15). Also, one can see that there is only one fail silence violation, which indicates that most of manifested heap errors are either caught by validity checks embedded in the JVM (15%) or result in segmentation violation of the JVM process (83%). On the contrary, the text-injection results show a larger manifested/injected ratio (e.g., 1134/2055=0.55 for libjava.so injections) and, more importantly, a significantly larger number of fail silence violations (3–6% of manifested errors). The latter result is due to errors in the program execution that are not captured by JVM internal checks.

Bytecode Error Injections. The bytecode-injection results of Table 10.3 show surprisingly large percentages of detected errors (71%), but a worrying amount of fail silence violations (26%). While the former figure indicates good coverage of the self-checking mechanisms within the code, the latter one is unacceptable if one wants to implement highly dependable systems. In most of the observed fail silence violations all non-injected processes block indefinitely and cannot recover from a single injected error.

Fail Silence Violations. Analysis of the fail silence violations reported in Table 10.3 reveals three main failure scenarios, reported in Table 10.4 and discussed below:

- *Liveness Violations*. An injected error propagates to non-injected processes, which eventually crash. As a result, the entire system goes down.
- *Uniform Safety Violations*. The injected replica violates the safety properties of atomic broadcast (e.g., a message lost or delivered out of order), but it occurs only at the injected replica.[2]
- *General Safety Violations*. An injected error propagates to non-injected processes, which violate the correctness properties of atomic broadcast without/before crashing.

Detailed analysis of the fail silence violation cases reveals the presence of reliability bottlenecks in Fortika's design, which is discussed in the next section.

Table 10.4: Breakdown of fail silence violations.

	JVM	JVM Text*			Bytecode
	Heap	libjava.so	libjvm.so	libnet.so	Stack*
Liveness violations					
1. Non-injected process hang	1	54 (84%)	5 (29%)	22 (73%)	448 (85%)
2. Non-injected process crash	0	3 (5%)	9 (53%)	8 (27%)	74 (14%)
Uniform safety violations	0	4 (6%)	3 (18%)	0	4 (0.8%)
General safety violations	0	3 (5%)	0	0	0
Total	1	64	17	30	526

* All percentages are with respect to the Total row.

10.3.5 Discovered Reliability Bottlenecks

This section discusses two key reliability bottlenecks we discovered in Fortika's design while performing an analysis of the fail silence violation cases of Table 10.3 and Table 10.4.

Partial Process Crashes. The largest contribution to the observed fail silence violations is shown in row 1 of Table 10.4 and corresponds to cases in which a single injected error causes the whole system to hang. Analysis of traces from these experiments reveals the following failure pattern. A thread t in the injected process p raises a Java Runtime exception. Because t's code does not catch such an exception, the JVM kills the offending thread t. However, other threads of process p are allowed to continue their execution normally. In the considered experiments, thread t happens to be a thread that executes a Fortika communication protocol

[2] With non-uniform properties this would not constitute a safety violation.

other than the Failure Detector protocol. Consequently, a non-injected process q never suspects the faulty process p, since p is still able to respond to ping messages. Nonetheless, all processes in the group eventually hang as they expect messages to be received from faulty process p, messages that should have been sent by the killed thread t.[3]

In summary, the observed fail silence violations are due to *partial* crashes of the injected process p, i.e., cases in which a critical communication thread is terminated while the thread responsible for responding to ping messages continues executing. Note that this behavior does not regularly occur in UNIX multithreaded applications, where the default operating system behavior is to terminate the whole multithreaded process if a thread performs an offending action. However, Java's default behavior is to terminate only the offending thread. This creates subtle problems when using multithreading in systems designed under the crash failure assumption. Overriding Java's default behavior to handle faulty threads is possible, yet it is made relatively simple only in the recently shipped Java 1.5.

Malformed Message Propagation. The second largest contribution to the observed fail silence violations is shown in row 2 of Table 10.4 and corresponds to cases in which a single injected error causes the whole system to crash. Our analysis indicates that the injected error results in invalid computation leading to a corrupted message being sent from the target process to some other processes. The recipient process raises an exception when reading the malformed message and, eventually, terminates in the exception handler.

Eliminating these error propagations requires changes to Fortika's design. Two cases are considered: (1) If a malformed message is received from an unreliable channel (e.g., an UDP socket), a recipient process can simply drop the message. As the channel is unreliable, the sender process will eventually retransmit the message because it does not receive an acknowledgment from the recipient. (2) If a malformed message is received from a reliable channel (e.g., a TCP socket), dropping the message does not suffice. TCP sockets handle acknowledgment messages transparently to higher layers and, thus, before Fortika code is delivered the message. The solution we propose in this case is to close the socket and suspect the sender process.

10.3.6 Assessment of Enhanced Fortika Design

This section presents a new set of injection experiments performed on Fortika enhanced by incorporating the design improvements discussed in Sect. 10.3.5. In the new design, we derive a new Java class from the standard Java class ThreadGroup, which groups together one or more Java threads. ThreadGroup allows us to override the default treatment of uncaught Java exceptions. Also, we have inserted code to handle reception of malformed messages from the network. As suggested in

[3]This is usually called *send omission* failures.

Sect. 10.3.5, we cope with a malformed UDP datagram by discarding the message, while in the case of a TCP stream we close the TCP socket and suspect the sender process. The changes made to the original code are small (approximately 20 lines of Java code). The results from the new experiments are reported in Table 10.5. One can clearly see that the number of fail silence violations is substantially reduced (e.g., 6% for Bytecode Stack injections against 26% in Table 10.3), while the number of internally detected errors (ASSERT column) is increased (e.g., 31% for libnet.so against 22% in Table 10.3). These results demonstrate the reliability benefit of the considered design improvements.

Table 10.5: Memory injection results for enhanced Fortika design.

Error Model	Total Injected Errors	Total* Activated Errors	Total† Manifested Errors	Manifested Errors‡			
				Crash Failures			Fail-silence violations
				SIGNAL	ASSERT	HANG	
JVM Heap	15177	N.A.	1221	1077 (88%)	123 (10%)	21 (2%)	0
JVM Text							
libjava.so	1000	779 (78%)	616 (79%)	425 (69%)	115 (18.7%)	40 (6.5%)	36 (5.8%)
libjvm.so	910	666 (73%)	269 (40%)	202 (75%)	45 (16.8%)	15 (5.6%)	7 (2.6%)
libnet.so	755	283 (37%)	215 (76%)	133 (62%)	67 (31%)	13 (6%)	2 (1%)
Bytecode Stack	5509	3049 (55%)	1825 (60%)	15 (0.8%)	1651 (91%)	50 (2.7%)	109 (6.0%)

* The ratio activated/injected is shown in parenthesis.
† The ratio manifested/activated is shown in parenthesis.
‡ Percentages with respect to total manifestations are shown in parentheses.

10.4 Network Injections

This section discusses the impact of errors in the messages exchanged by Fortika processes. A single-bit error is injected in a randomly selected message sent by a target process. The corrupted message propagates to another process and can possibly cause a failure at the receiving end. In this way, we mimic failures occurring in the system and leading to malformed messages being sent/received. Note that the modeled errors occur before/after any encoding (e.g. checksum) is applied/removed to protect messages against transmission errors. The goals are: (1) to test the robustness of Fortika with respect to erroneous input data (e.g. corrupted network messages) and (2) to provide further insight into the error propagation mechanisms leading to the fail silence violations observed in the memory injection experiments.

In order to perform the network injection experiments, we have extended the NFTAPE framework [SFKI00] to include a network-error injector customized for the Fortika framework. The injection mechanism used is similar to the one proposed in [DJMT96]. By inserting an additional layer between the Java Virtual Machine (on which Fortika code executes) and the standard socket interface of the operating system, we can alter the messages that pass through the layer.[4] This

[4]In practice, the injector is in a shared library that overloads, at run-time, the operating system's shared library that offers the socket interface.

Table 10.6: Fortika message types.

Message Type	Description
Length	Short messages of fixed length (4 bytes) that are used by Fortika's Reliable Channel protocol module to specify the length of a subsequent message of variable size.
Abcast	Messages used by Fortika's Atomic Broadcast protocol module to convey an application message from the broadcast sender to each destination.
Propose	Messages used by Fortika's Consensus protocol module to convey the proposed value that the coordinator tries to impose as the decision. Only the coordinator sends these messages [CT96].
Ack	Messages used by the Consensus protocol module to acknowledge the reception of a *propose* message. Acknowledgment messages contain a boolean variable that specifies whether the message is a positive acknowledgment (*ack*) or a negative acknowledgment (*nack*) [CT96]. Since the coordinator never sends these messages, injection experiments for *ack* messages target a non-coordinator process.
Decide	Messages used by the Consensus protocol module to convey the final decision of an instance of consensus to all processes [CT96].
Alive	UDP messages used by Fortika's Failure Detector protocol module to ping alive processes.
Other	Other message types used by Fortika's modules. They are rarely used (0.1%, see Figure 10.3) and we did not obtain any interesting result injecting those messages.

allows us to monitor messages exchanged by Fortika processes with minimal intrusiveness. Besides injecting errors, the network-error injector enables reliably logging of experiment data to persistent storage for off-line analysis.

10.4.1 Message Types

This section introduces Fortika message types. The discussion is essential in analyzing and explaining the observed system behavior. Figure 10.3 shows how frequently Fortika message types are used during an experimental run (the data is obtained by profiling network activity during a single experiment). Table 10.6 provides a concise description of the six message types that are most frequently used.[5]

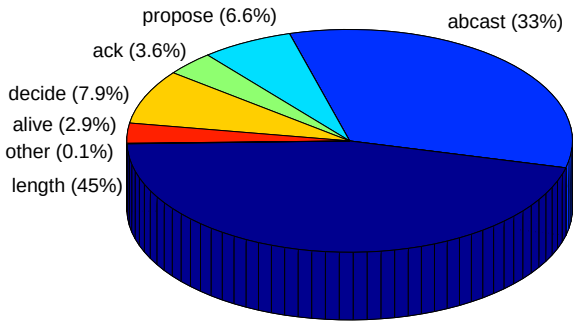


Figure 10.3: Fortika message profiling.

In order to illustrate how these messages are used by Fortika's Atomic Broadcast and Consensus protocol modules, Figure 10.4 depicts an execution scenario

[5]Unless explicitly stated, these messages are transmitted over TCP connections.

where a non-coordinator process r atomic-broadcasts a message m to the group members. In the figure, horizontal arrows represent time, oblique arrows represent message exchanges, abcast(m) indicates that the application (benchmark) has executed abcast of message m, and adeliver(m) indicates the adelivery of message m to the application.

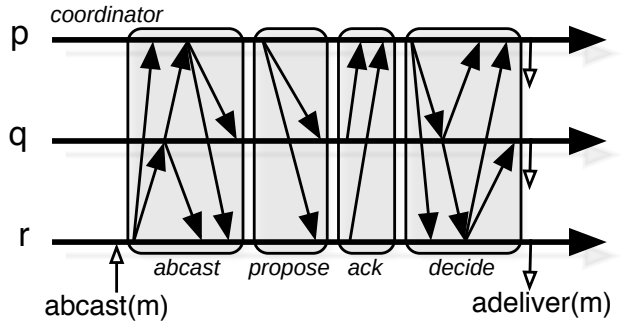


Figure 10.4: Atomic Broadcast execution example.

10.4.2 Error Models and Outcome Categories

The network injection campaign is performed by corrupting a random bit in a message sent by the target process. The injected message is randomly selected from the messages exchanged during a single experiment, i.e. a single run of the benchmark application. A separate set of experiments is performed to corrupt each message type introduced in Sect. 10.4.1. The outcome categories considered are based on those introduced in Sect. 10.3.1. Since a network injection occurs when a target process sends a message to another group member, all failure scenarios defined in Sect. 10.3.1 correspond to the failure of a recipient of the corrupted message. Additionally, we distinguish between the following failure scenarios:

Unmarshal exception. The recipient process raises a Java exception because it is unable to unmarshal the corrupted message received from the network.

Incorrect unmarshal. The recipient process incorrectly unmarshals the corrupted message, which leads to the allocation of invalid objects in memory. The recipient process can either (i) immediately crash while trying to access the invalid objects (*no propagation* case), or (ii) behave erratically by sending an invalid message to other processes (*propagation* case).

10.4.3 Network Injection Results

Table 10.7 reports the results from the network injections. We distinguish between not marshaled messages (*length* messages), which do not use standard Java Serial-

Table 10.7: Network injection results.

Message type	Total experiments	Total manifested errors*	Manifested errors†		
			Unmarshal exception	Incorrect unmarshal	
				No propagation	Propagation
Not marshaled					
length	961	831 (86.5%)‡	N.A.	N.A.	N.A.
Marshalled					
abcast	996	658 (66.1%)	560 (85.1%)	92 (14.0%)	6 (0.9%)
propose	987	641 (64.9%)	555 (86.6%)	86 (13.4%)	0
ack	995	851 (85.5%)	633 (74.4%)	218 (25.6%)	0
decide	996	598 (60.0%)	528 (88.3%)	69 (11.5%)	1 (0.2%)
alive	999	671 (67.2%)	604 (90.0%)	67 (10.0%)	0

* Percentages with respect to total experiments are shown in parentheses.
† Percentages with respect to total manifestations are shown in parentheses.
‡ In all the cases the receiving process terminates after raising an OutOfMemoryError Java exception.

ization, and marshaled messages (the rest of the messages). The following observations can be made:

- Less-than-100% observed manifestation rate (i.e., percentage between manifested and injected errors) can be explained by the redundant information that marshaled Java objects carry (e.g., full description of ancestor classes). This information is not used by the unmarshal routines unless sender and receiver processes have different versions of the Java classes that are serialized in the network messages, which is not the case in our setup.

- In a significant number of cases a corrupted message is incorrectly unmarshaled, with subsequent memory corruption in the recipient process and possibly erratic behavior of the process. In particular, injections in *ack* messages show 218 of such cases. Detailed analysis of the results reveals that these scenarios are due to flaws in the mechanisms (1) implemented by Fortika to manage event-routing information, and (2) implemented by standard Java Serialization to transmit Java objects over the network. These two reliability bottlenecks are discussed in detail in Sect. 10.4.4.

- Error propagations are observed only for *abcast* and *decide* messages. Detailed analysis reveals that these propagations always follow the same pattern: (1) a corrupted message m is incorrectly unmarshaled and reaches either the Consensus or the Atomic Broadcast module of the recipient process; (2) as the injected error corrupted some part of the message that the module does not process (e.g., message headers used by an upper-level protocol module such as the Membership module), the recipient module can resend m to all other processes—accordingly with the module's algorithm—before delivering m to the upper level modules; (3) m is received by all group members, which eventually fail when they process the corrupted portion of m.

Table 10.8: Breakdown of experiments with incorrectly unmarshaled messages in Table 10.7.

Affected message region	Message type*				
	abcast	*propose*	*ack*	*decide*	*alive*
1. Fortika event-routing data	68 (69.4%)	49 (70.0%)	157 (72.0%)	57 (66.3%)	0
2. Java Serialization					
2.1. Ancestor classname	22 (22.4%)	19 (27.1%)	61 (28.0%)	27 (31.4%)	59 (88.1%)
2.2. Attribute name	2 (2.0%)	0	0	0	6 (9.0%)
3. Protocol data	6 (6.1%)	2 (2.9%)	0	2 (%)	2 (3.0%)
Total	98	70	218	86	67

* Percentages with respect to total row are shown in parentheses.

10.4.4 Discovered Reliability Bottlenecks

The performed network injection experiments enabled us to discover two important reliability bottlenecks. The first is related to connectors: the mechanism implemented by Fortika to route events to the correct protocol modules (see Sect. 3.4.2). The second is related to the marshaling/unmarshaling mechanisms implemented by the standard Java Serialization library [Sun01]. Both cases result in a corrupted message being incorrectly unmarshaled (see Table 10.7). Table 10.8 provides a breakdown of these cases.

Fortika Event Routing Mechanism. Connectors are implemented in Fortika in the following manner. Messages contain a *classname* string that indicates the Java class of the protocol module to which the message is addressed. An error injected in this classname string results in the incorrect unmarshaling of a corrupted message. Row 1 of Table 10.8 shows that 60–70% of incorrectly unmarshaled cases are due to such error cases. Because of a design flaw, when the receiver process unmarshals a corrupted classname string (e.g., a string indicating a non-existent Fortika protocol module) no error is detected by Fortika, and the message is routed to an arbitrary protocol module. We corrected this problem by using integer IDs instead of classname strings to indicate a destination module, and by checking for the existence of a protocol module indicated by received messages. If the check fails, we use the same policy proposed in Section 10.3.5: the receiving process either (1) drops the invalid message (if the message is sent over UDP), or (2) closes the socket and suspects the message sender (if the message is sent over TCP).

Table 10.9: Network injection results with the new design.

Message type	Total experiments	Total manifested errors*	Manifested errors†		
			Unmarshal exception	Incorrect Unmarshal	
				No propagation	Propagation
Not marshaled	829	734 (88.5%)‡	N.A.	N.A.	N.A.
Marshalled	1062	625 (58.9%)	543 (86.8%)	76 (12.1%)	6 (1.1%)

* Percentages with respect to total experiments are shown in parentheses.
† Percentages with respect to total manifestations are shown in parentheses.
‡ In all the cases the receiving process terminates after raising an OutOfMemoryError Java exception.

Java Serialization Mechanism. Fortika makes intensive use of Sun's Java Serialization mechanisms (see Sect. 10.3.3), which is highly stressed by our network injections. In the experiments reported in rows 2.1 and 2.2 of Table 10.8, Java Serialization mechanisms fail to detect corruption in a serialized message, which results in an incorrect unmarshaling of the corrupted message. Analysis of the experimental traces shows that the problem is due to two subtle issues in the Java Serialization code:

- If the injected error affects the name of an ancestor class (row 2.1 in Table 10.8), the unmarshaling routines return a null Java object instead of throwing an exception to report an unknown Java class.
- If the injected error affects the name of an attribute in a class description (row 2.2 in Table 10.8), the unmarshaling routines create an attribute with a default value (e.g., 0 for integer attributes) instead of throwing an exception to indicate an unknown attribute name.

The data in Table 10.8 shows that errors affecting classnames are more frequent (row 2.1) than errors affecting attribute names (row 2.2). The reason can be attributed to the longer length of a full classname with respect to an attribute name.

To remove the two problems described above, the Java unmarshaling code must be modified. This task, however, is not as simple as it may seem. First, a possible reason for the observed behavior of Java Serialization code is that it is designed to support compatibility between different versions of the same Java class. Thus, changing the code to throw an exception instead of replacing an invalid field with a default value may break this compatibility. Second, modifying Sun's JDK code is not permitted by Sun's end user license agreement.

10.4.5 Assessment of Enhanced Fortika Design

This section presents a new set of injection experiments performed on Fortika enhanced by incorporating the design improvements discussed in Sect. 10.4.4 to improve Fortika's mechanism to route events through protocol modules. The results from the new set of network injections are summarized in Table 10.9. One can see that the percentage of incorrectly unmarshaled messages is lower than the one observed in the previous experiments of Table 10.7, yet it is not null. The reason for this can be attributed to the error contribution brought by the Java Serialization code (see Sect. 10.4.4), which is left unchanged from the experiments reported in Table 10.9 (see Sect. 10.4.4).

10.5 Java *vs.* OCAML

Our initial goal was to use error injection to compare Fortika's error resilience, discussed in this chapter, with that of Ensemble, studied using error injection in [BWKI03]. An important finding from [BWKI03] is that despite a significant

effort spent in formally modeling and proving the correctness of the Ensemble design [KHH98], the Ensemble implementation exhibits high sensitivity to real errors (in memory segments and in network messages), which often leads to uncontrolled crashes of the entire system and creates significant impediment in achieving high dependability. This work demonstrates how the lessons learned from the Ensemble study and the use of error injection as an evaluation methodology can be used to guide a designer to enhance a system (Fortika in our case) so that it achieves a desired level of reliability or availability. While carrying out this work, we soon realized that the error resilience of the systems under study (Fortika and Ensemble) are greatly influenced by the platform on which the systems are built. [BWKI03] states that Ensemble layers (i.e., protocol modules) constitute only 5% of the runtime function invocations, while a significant percentage (50%) of invocations is for the runtime language support of OCAML: the functional language in which Ensemble is written. In Sect. 10.3.3 we show that the Java Virtual Machine and the associated Java libraries account for 98% of the runtime function invocations in Fortika. These observations are fundamental when building a fault-tolerant system, because they show that a major source of reliability (or unreliability) is in parts of the system (the underlying OCAML or Java platform) over which the designer has no (or little) control.

Table 10.10: Comparison of Fortika and Ensemble memory injection results.

	Error Model	Total Injected Errors	Total* Activated Errors	Total† Manifested Errors	Manifested Errors			
					Crash Failures			Fail-silence violations
					SIGNAL	ASSERT	HANG	
OCAML (Ensemble)	Heap	11278	N.A.	412	352 (85%)	26 (6%)	13 (3%)	21 (5%)
	Text	7401	2583	1878 (73%)	1604 (85%)	142 (8%)	27 (1%)	105 (6%)
Java (Fortika)	JVM Heap	15177	N.A.	1221	1077 (88%)	123 (10%)	21 (2%)	0
	JVM Text							
	libjava.so	1000	779 (78%)	616 (79%)	425 (69%)	115 (19%)	40 (7%)	36 (6%)
	libjvm.so	910	666 (73%)	269 (40%)	202 (75%)	45 (17%)	15 (6%)	7 (3%)
	libnet.so	755	283 (37%)	215 (76%)	133 (62%)	67 (31%)	13 (6%)	2 (1%)

* The ratio activated/injected is shown in parenthesis.
† The ratio manifested/activated is shown in parenthesis.

Table 10.11: Comparison of Fortika and Ensemble network injection results.

	Injected experiments	Manifested† errors	Manifested errors		
			SIGNAL	ASSERT	Erratic Beh.
Serialized OCAML Messages (Ensemble)	999	189 (19%)	124 (66%)	11 (6%)	44 (28%)
Serialized Java Messages (Fortika)	1062	625 (59%)	0	543 (87%)	82 (13%)

† The ratio manifested/injected is shown in parenthesis.

To illustrate these aspects, Table 10.10 and Table 10.11 summarize the assessment results obtained for Fortika and Ensemble,[6] in the case of errors in memory

[6]The data considered for Ensemble is from the experiments with Ensemble's sequencer-based

segments and in network messages, respectively. The following observations can be made:

- Table 10.10 shows that Java does a better job in providing crash-failure semantics to applications. The coverage of internal self-checking mechanisms (ASSERT column) and the percentage of fail silence violations are noticeably improved with respect to OCAML. Note that here we are considering Fortika executed as interpreted Java bytecode and Ensemble compiled to native machine-code. Hence, there remains the question of whether a Java application preserves this improved reliability when compiled in a native machine-code (e.g., by the Java just-in-time compiler).

- Table 10.11 shows a three-fold increase in the percentage of manifested errors when using Java. This result can be attributed to more compact serialized messages for Java than for OCAML. When an error does manifest, however, Java is significantly more efficient in detecting it (ASSERT column), and preventing uncontrolled crashes (SIGNAL column) and erratic behavior (last column).

To summarize, it appears that Java provides a better development platform for fault-tolerant systems than OCAML. This result is quite unexpected, and perhaps controversial, given that OCAML applications are more amenable for formal analysis [KHH98]. Nonetheless, recall the serious deficiencies, for both reliability and performance, of the standard Java Serialization mechanisms discussed in Section 10.4.4.

10.6 Conclusion

This chapter provides a thorough study of error effects in Fortika and demonstrates how experimental error injection can be integrated in the design process of a robust fault-tolerant system. Use of error injection enables us to uncover subtle reliability bottlenecks both in the design of Fortika and in the implementation of Java. One of the results obtained in Chapter 3 is that standard Java Serialization is a performance bottleneck for Fortika. In this chapter, error injection in Fortika shows that standard Java Serialization is also a serious dependability bottleneck. Given the emerging popularity of Java-based fault tolerance middleware, the significance of our findings is clear. Comparison of Fortika's error assessment results with those obtained for the Ensemble toolkit in [BWKI03] allows us to investigate the reliability implications that the choice of the development platform can have when building fault-tolerant systems. We argue that Java provides a better enforcement of the crash-failure semantics than OCAML, the functional language in which Ensemble is written. This result is unexpected, given that OCAML applications are more amenable for formal analysis [KHH98].

atomic broadcast protocol.

Chapter 11

Conclusion

11.1 Research Assessment

In Part I, we analyzed state-of-the-art protocol composition frameworks and proposed novel features to enhance their design. In Part II, we studied modular group communication protocols in the context of several system models. We proposed a new architecture to structure protocol modules in the dynamic system model, and we specified and implemented atomic broadcast in the crash-recovery system model. In Part III, we explain how to combine our contributions in the fields of protocol composition frameworks and modular group communication in order to build a new modular group communication toolkit: Fortika. Also in Part III, we put Fortika robustness under test using software-based fault injections. We now assess each of the contributions in more detail.

11.1.1 Protocol Composition Frameworks

Perspectives for Framework Description and Comparison. We proposed four perspectives for framework description and comparison, namely (1) the *composition model*, which determines how protocol modules are arranged in a composition, (2) the *interaction model*, which specifies how protocols modules can communicate within the local process, (3) the *concurrency model*, which defines how concurrency is managed, and (4) the *interface with the environment*, which describes how the protocol modules can interact with the code outside the composition.

We used these four perspectives systematically whenever protocol composition frameworks were described or compared. Moreover, they constitute the backbone of Part I's structure: Chapter 3 explored all four perspectives, Chapter 4 focused on the concurrency model and some aspects of the interface with the environment, and Chapter 5 proposed a new interaction model along with a composition model.

Comparing two Frameworks Using the Four Perspectives Defined. We compared Appia and Cactus, two of the most relevant protocol composition frame-

works. The preliminary result was that neither framework was neatly better than the other. However, a closer comparison using the four perspectives presented above, yielded the following conclusions. (1) Cactus's non-hierarchical composition model is more flexible than the strict order imposed by Appia's *channels*. (2) As for the interaction model, none of the frameworks provides a convincing mechanism to multiplex events transparently to protocol modules. (3) Appia's single-threaded concurrency model is easier to use and less error-prone than the unrestricted concurrency offered by Cactus. (4) Appia offers a good mechanism for communication from the environment to the composition but a similar mechanism is missing for communication from the composition to the environment.

We also measured the performance of Appia and Cactus, the latter being significantly faster in our tests. To investigate these results we performed an execution profiling of the tests, which uncovered the performance bottleneck in both frameworks: standard Java serialization. The execution time spent on serializing objects was particularly long in Appia, which explains why it performed slower.

On the Concurrency of Protocol Composition Frameworks. We surveyed the concurrency model of eight protocol composition frameworks, analyzing the way they provide concurrency and the means they offer to deal with multiple threads. We proposed a set of features that these frameworks can offer with a negligible performance cost: (1) islands of protocol modules with reactive behavior that can coexist with active protocol modules; (2) non-overlapping execution of protocol modules involved in a chain of events, to avoid inconsistencies; and (3) *extended causal order* as the best ordering guarantee for handler invocation. We provide a definition of extended causal order, which is the simplest we are aware of.

The Header-Driven Interaction Model. Events are present in most protocol composition frameworks nowadays. The main reason is that they reflect the reactive nature of protocols in an intuitive way. However, we showed that events are not as unquestionable as they may seem: we pointed out inherent drawbacks and presented a new model for protocol interaction: the header-driven model. Its main advantages are (1) better routing mechanisms, (2) better type information in messages, (3) better protocol readability, and (4) less compositional problems.

The header-driven model does not have events at the core of its interaction scheme: it uses message headers instead. For this reason, legacy event-driven protocols are not compatible with header-driven ones. This is the main reason why Fortika (as it is at present) does not support event-driven protocols.

11.1.2 Modular Group Communication

A New Architecture for Modular Group Communication. We proposed a new architecture for group communication toolkits in the dynamic system model, i.e., providing a group membership service. The architecture proposed has the following advantages: (1) it is less complex: all ordering problems are solved at the same

place; (2) it is more powerful: not only can it be used for both passive and active replication, but also for other customized setups; and (3) it can be made more responsive to failures thanks to its double timeout scheme: very short timeout to avoid blocking and a very long one to eventually exclude crashed processes.

Atomic Broadcast in the Crash-Recovery Model. We discussed why existing specifications of atomic broadcast in the crash-recovery model are not satisfactory; and proposed two new specifications: a uniform and a non-uniform one. The concept of uniformity is not well understood in the crash-recovery model and we defined it as an extension of the concept of uniformity in the crash-stop model, which is widely accepted. We proposed the distinction between permanent and volatile events as the key to define uniform and non-uniform properties. We then presented implementations for uniform and non-uniform atomic broadcast. Our non-uniform atomic broadcast algorithm does not require frequent disk accesses, and has thereby the potential to achieve a level of performance comparable to atomic broadcast algorithms in the crash-stop model, which never access the disk.

11.1.3 Fortika

The *Fortika* Group Communication Toolkit. Fortika is the main prototype implementation in the context of the thesis. It is a novel group communication toolkit, written in Java, whose algorithmic code is not targeted to the protocol composition framework used to build the composition. This is achieved by a set of conventions that protocol programmers must observe. As a result, Fortika's protocol modules can be composed using any event-driven framework. Fortika has been the testbed of the architectures and algorithms studied in the thesis. Currently, it includes compositions offering atomic broadcast for three different system models: (1) the static crash-stop model, used to compare Appia and Cactus, (2) the dynamic crash-stop model, which demonstrates the new architecture proposed for modular group communication, and (3) the static crash-recovery model, which is a proof-of-concept implementation of the atomic broadcast algorithms proposed.

Assessing the Crash-Failure Assumption with Fault Injection. Fortika protocols (as most group communication toolkits) assumes that processes can only fail by crashing. We used software-based fault injectors to test Fortika's robustness in the presence of more severe failure types like spurious memory or message corruption. We uncovered several reliability bottlenecks both in Fortika and in Java. Fortika's design was enhanced to properly transform the errors injected into clean crash failures. Finally, our results were compared with a previous work that performed fault injection into Ensemble. The (perhaps controversial) conclusion of this comparison was that the Java platform offers better support than OCAML (the functional language in which Ensemble is written) to cleanly transform memory and message corruption into clean crash failures.

11.2 Open Questions and Future Research Directions

Fortika Protocol Modules and the Header-Driven Model. Fortika conventions for protocol programmers assume that protocol modules use events to interact. This is the key issue that makes it possible to use any event-driven protocol composition framework in Fortika. On the other hand, a contribution of this thesis is the header-driven interaction model presented in Chapter 5, which drops events and uses messages headers exclusively. As a future work we envisage to adapt protocols in Fortika so that they can work with message headers. We reckon that the adaptation will be deep in the sense that it will require a substantial rewriting of protocol modules, which will probably become incompatible with event-driven protocol composition frameworks.

Dynamic Generic Broadcast. Our new architecture for group communication toolkits in the dynamic system model includes generic broadcast as an important protocol module. The current implementation in Fortika does not yet have generic broadcast. Even though some implementations of generic broadcast have been proposed in the literature, all of them assume a *static* system model, and not a *dynamic* one as needed here. This is the main obstacle to including a generic broadcast protocol module. We are currently studying existent implementations of generic broadcast in the static model in order to come up with a dynamic implementation that can be incorporated to Fortika's dynamic composition.

Improving Specification of Consensus in the Crash-Recovery Model. We have introduced the *commit* primitive in Chapter 8, which is instrumental to a clean distinction between volatile and permanent events. This is the main innovation of our specification of atomic broadcast. After a preliminary study, we believe that consensus specification in the crash-recovery model can be considerably improved if it takes into account permanent and volatile events as well. We envision this as future work.

Combining the Crash-Recovery and Dynamic Models. We have conducted our research in the (1) static crash-stop, (2) dynamic crash-stop, and (3) static crash-recovery system models. The obvious missing combination is the *dynamic crash-recovery* model. In such a model, not only could processes be added and removed on the fly (group membership), but also crashed processes could recover and, thereby, would not necessarily be excluded from the group. This would avoid costly state transfer operations every time a crashed process is replaced by another one (which could be a different incarnation of the same process). We plan to assess the interest of this system model, as well as propose specifications and implementations for this model.

Optimization of Fortika. We have recently modified Fortika so that it does not use standard Java serialization, but a light-weight marshaling library [HMP05]. Instead of fully describing the class to be serialized and all its ancestors, as standard Java serialization does [Sun01], this library only marshals the class name and the attribute values. Preliminary measurements indicate that the gain in performance is significant. As a result, it appears that serialization is no more the performance bottleneck in Fortika; we intend to run again Fortika with an execution profiler to detect new performance bottlenecks. The ultimate goal along this path is to optimize Fortika to a maximum extent.

Fault Injection with the New Serialization Library. In Chapter 10, we saw that standard Java serialization is also a serious reliability bottleneck since its unmarshaling routines fail to detect errors injected in certain parts of a message. A natural follow-up of this work is to assess and compare the robustness of the new light-weight serialization library [HMP05] under the same error conditions.

Bibliography

[AAEAS97] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, number 1300 in Lecture Notes in Computer Science, pages 496–503, Passau, Germany, August 1997. Extended abstract.

[ACT00] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[ADGFT00] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In M. Herlihy, editor, *Proc. 14th Int'l Symp. on Distributed Computing (DISC'00)*, volume 1914 of *LNCS*, pages 268–282, Toledo, Spain, October 2000.

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

[AR01] M. Achour and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11:95–107, 2001.

[AT02] Y. Amir and C. Tutu. From total order to database replication. In *Proc. 22nd IEEE Intl. Conf. on Distributed Computing Systems (ICDCS-22)*, pages 494–503, Vienna, Austria, July 2002.

[Ban02] B. Ban. *JavaGroups 2.0 User's Guide*, Nov 2002.

[Bar81] J. F. Bartlett. A NonStop kernel. In *Proceedings of the 8th ACM Symp. on Operating Systems Principles (SoSP-8)*, November 1981.

[BDGB95] O. Babaoglu, R. Davoli, L. Giachini, and M. Baker. Relacs: A communication infrastructure for constructing reliable applications in large-scale distributed systems. In *Proceedings of the 28th Hawaii Interntional Conference on System Sciences*, volume II, pages 612–621, Jan 1995.

[BG00] R. Boichat and R. Guerraoui. Reliable broadcast in the crash-recovery model. In *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nuremberg, Germany, oct 2000.

[BGH87] J. Bartlett, J. Gray, and B. Horst. *The Evolution of Fault-Tolerant Systems*, chapter Fault Tolerance in Tandem Computer Systems, pages 55–76. Springer-Verlag, 1987.

[BGT+01] F. Brasileiro, F. Greve, F. Tronel, M. Hurfin, and J.-P. Le Narzul. Eva: An event-based framework for developing specialized com munication protocols. In *Proc. IEEE Int'l Symp. on Network Computing and Applications (NCA'01)*, pages 108–119, Cambridge, MA, USA, October 2001.

[Bha96] N. T. Bhatti. *A system for constructing configurable high-level protocols*. PhD thesis, University of Arizona, 1996.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. http://research.microsoft.com/pubs/ccontrol/.

[BHS98] N. T. Bhatti, M. A. Hiltunen, and D. Schlichting. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–366, November 1998.

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

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

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

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

[BvR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[BWKI03] C. Basile, L. Wang, Z. Kalbarczyk, and R. Iyer. Group communication protocols under errors. In *Proc. of 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, 2003.

[CBDS02] B. Charron-Bost, X. Défago, and A. Schiper. Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 244–249, Osaka, Japan, October 2002.

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

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

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

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

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

[CT91] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC-10)*, pages 325–340, Montreal, Quebec, Canada, August 1991.

[CT94] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report TR94-1458, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, October 1994.

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

[CZ85] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Trans. on Computer Systems*, 3(2):77–107, May 1985.

[DFS00] L. Duchien, G. Florin, and L. Seinturier. Partial order relations in distributed object environments. *SIGOPS Oper. Syst. Rev.*, 34(4):56–75, 2000.

[DJMT96] S. Dawson, F. Jahanian, T. Mitton, and T. Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proc. of the Symp. on Fault-Tolerant Computing*, pages 404–414, 1996.

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

[DM96] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Comm. ACM*, 39(4):64–70, April 1996.

[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.

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

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-Oriented Language For Communicating System s*. Prentrice Hall, Harlow, England, 1997.

[EMPS04a] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards flexible finite-state-machine-based protocol composition. Technical report, IC/2004/63. École Polytechnique Fédérale de Lausanne, July 2004.

[EMPS04b] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards flexible finite-state-machine-based protocol composition. In *Proceedings of the 3rd International Symposium on Network Computing and Applications (IEEE NCA04)*, Cambridge, MA, USA, August 2004.

[EMS95] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS-15)*, pages 296–306, Vancouver, Canada, May 1995.

[ES06] R. Ekwall and A. Schiper. Solving atomic broadcast with indirect consensus. In *2006 IEEE International Conference on Dependable Systems and Networks (DSN 2006)*, 2006.

[EUS02] R. Ekwall, P. Urbán, and A. Schiper. Robust TCP connections for fault tolerant computing. In *Proc. 9th Int'l Conf. on Parallel and Distributed Systems (ICPADS)*, Chung-li, Taiwan, December 2002.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical Report 273, Department of

Computer Science, Yale University, New Haven, Conn., USA, June 1983.

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

[Fuc98] E. Fuchs. Validating the fail-silence assumption of the MARS architecture. In *Proc. of the Dependable Computing for Critical Applications Conf.*, pages 225–247, 1998.

[FvR96] R. Friedman and R. van Renesse. Strong and Weak Virual Synchrony in Horus. In *15th IEEE Symp. on Reliable Distributed Systems (SRDS-15)*, pages 140–149, Niagara-on-the-Lake, Ontario, Canada, September 1996.

[Gen04] C. Gensoul. Implementing nuntius in the Objective Caml system. Master's thesis, École Polytechnique Fédérale de Lausanne (EPFL), 2004.

[GOS98] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical Report 98/272, École Polytechnique Fédérale de Lausanne, Switzerland, March 1998.

[GP96] A. Galleni and D. Powell. Consensus and Membership in Synchronous and Asynchronous Distributed Systems. RR 96104, LAAS-CNRS, Toulouse, April 96.

[Gra86] J. Gray. Why do computers stop and what can be done about it ? In *Proceedings of the 5th Symposium on Reliablity in Distributed Software and Database systems*, January 1986.

[GS97] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[Hay98] M. Hayden. The Ensemble system. Technical Report TR98-1662, Dept. of Computer Science, Cornell University, January 8, 1998.

[HMP05] B. Haumacher, T. Moschny, and M. Philippsen. *Fast Object Serialization:* `uka.transport`. Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, March 2005.

[HMR98] M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS)*, pages 280–286, West Lafayette, IN, USA, October 1998.

[HP91] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Trans. on Software Engineering*, 17(1):64–76, January 1991.

[HS98] M. A. Hiltunen and R. D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[HS00] M. A. Hiltunen and R. D. Schlichting. The Cactus approach to building configurable middleware services. In *Proc. Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nürnberg, Germany, October 2000.

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

[IBM00] IBM Corporation. *SockPerf: A Peer-to-Peer Socket Benchmark Used for Comparing and Measuring Java Socket Performance*, 2000.

[ISO96] ISO. *Information technology – Open Systems Interconnection – Connection-oriented Session protocol: Protocol specification*. ISO/IEC 8327-1. International Organization for Standards, 1996.

[JKN96] W. Jia, J. Kaiser, and E. Nett. RMP: Fault-tolerant group communication. *IEEE Micro*, 16(2):59–67, April 1996.

[KA00] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3):333–379, 2000.

[KE05] A. Kupsys and R. Ekwall. Architectural issues of JMS compliant group communication. In *4th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2005)*, 2005.

[Kem00] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis 13864, Swiss Federal Institute of Technology, Zürich, Switzerland, August 2000.

[KHH98] C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. *Lecture Notes in Computer Science*, 1421:317–332, 1998.

[KM00] J. S. I. Keidar and K. Marzullo. Optimistic virtual synchrony. In *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 42–51, Nuremberg, Germany, October 2000.

[KPSW04] A. Kupsys, S. Pleisch, A. Schiper, and M. Wiesmann. Towards JMS compliant group communication - a semantic mapping. In *Proceedings of the 3rd International Symposium on Network Computing and Applications (IEEE NCA04)*, 2004.

[KT91] F. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, pages 222–230, Arlington, TX, USA, May 1991.

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

[Lap92] J.-C. Laprie, editor. *Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese*, volume 5 of *Dependable Computing and Fault Tolerant Systems*. Springer-Verlag, 1992.

[Ler00] X. Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Progr. Languages and Syst.*, 4(3):382–401, 1982.

[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, September 1996.

[MDB01] A. Montresor, R. Davoli, and Ö. Babaoğlu. Middleware for dependable network services in partitionable distributed systems. *Operating Systems Review*, 35(1):73–84, January 2001.

[Men01] S. Mena. Configuration and extension of group communication protocols. Final Report. Graduate School in Computer Science. École Polytechnique Fédérale de Lausanne, July 2001.

[MFSW95] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA, October 1995. Workshop held during the 7th IEEE Symp. on Parallel and Distributed Processing, (SPDP-7).

[MJ94] H. Madeira and J.G.Silva. Experimental evaluation of the fail-silent behavior in computers without error masking. In *Proc. of the Int'l Symp. on Fault-Tolerant Computing*, pages 350–359, 1994.

[MMSA+96] L. E. Moser, P. M. Melliar-Smith, D. A. Agrawal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Comm. ACM*, 39(4):54–63, April 1996.

[Mon94] T. Montgomery. Design, implementation, and verification of the reliable multicast protocol. Master's thesis, West Virginia University, Dec 1994.

[MPR00] H. Miranda, A. Pinto, and L. Rodrigues. *Application Program Interface Specification of Appia*, November 2000.

[MPR01] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st Int'l Conf. on Distributed Computing Systems (ICDCS' 01)*, pages 707–710, Washington - Brussels - Tokyo, April16–19 2001.

[MPS93] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87–103, 1993.

[MR99a] H. Miranda and L. Rodrigues. Communication support for multiple QoS requirements. In *3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island, Portugal, April 1999.

[MR99b] H. Miranda and L. Rodrigues. Flexible communication support for CSCW applications. In *5th Int'l Workshop on Groupware - CRIWG'99*, pages 338–342, Cacún, México, September 1999. IEEE.

[OGS97] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report 97/239, École Polytechnique Fédérale de Lausanne, Switzerland, August 1997.

[PGS98] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, September 1998.

[Pin01] A. Pinto. Appia group communication manual, February 2001.

[PMJPKA00] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In M. Herlihy, editor, *Proc. 14th Intl. Symp. on Distributed Computing (DISC-14)*, volume 1914 of *LNCS*, pages 315–329, Toledo, Spain, October 2000.

[PMR01] A. Pinto, H. Miranda, and L. Rodrigues. Light-weight groups: an implementation in ensemble. In *Fourth European Research Seminar on Advances in Distributed Systems (ERSADS'01)*, Bertinoro (Forli),Italy, May 2001.

[PS99] F. Pedone and A. Schiper. Generic broadcast. Technical Report SSC/1999/012, École Polytechnique Fédérale de Lausanne, Switzerland, April 1999.

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

[Ric96] A. Ricciardi. Impossibility of (repeated) reliable broadcast. Technical Report TR-PDS-1996-003, Univ of Texas, Austin, April 1996.

[ROT94] M. Rimen, J. Ohlsson, and J. Torin. On microprocessor error behavior modeling. In *Proc. of the Int'l Symp. on Fault-Tolerant Computing*, 1994.

[RR03] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(4), 2003.

[RWS06] O. Rütti, P. T. Wojciechowski, and A. Schiper. Structural and algorithmic issues of dynamic protocol update. In *20th IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[Sch93] F. B. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 7, pages 169–198. Addison-Wesley, second edition, 1993.

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

[SFKI00] D. Stott, B. Floering, Z. Kalbarczyk, and R. Iyer. Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE. In *Proc. of the Int'l Computer Performance and Dependability Symp.*, 2000.

[SS93] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS-13)*, pages 561–568, Pittsburgh, Pennsylvania, USA, May 1993. IEEE Computer Society Press.

[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.

[Sun01] Sun Microsystems Inc., Palo Alto, CA. *Java Object Serialization Specification*, 2001. Revision 1.4.4.

[UDS02] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6):981–997, November 2002.

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

[vRBC+93] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus system. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, 1993.

[vRBG+96] R. van Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, April 1996.

[VRBM96] R. Van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Comm. ACM*, 39(4):76–83, April 1996.

[WHS01] G. Wong, M. Hiltunen, and R. Schlichting. CTP: A configurable and extensible transport protocol. In *Proceedings of the 20th Annual Conference of IEEE Communications and Computer Societies (INFOCOM 2001), Anchorage, Alaska*, April 2001.

[Wie02] M. Wiesmann. *Group Communications and Database Replication: Techniques, Issues and Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, May 2002. Number 2577.

[WMK94] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In Springer-Verlag, editor, *Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science, pages 33–57, Dagstuhl Castle, Germany, September 1994.

[WMS02a] P. T. Wojciechowski, S. Mena, and A. Schiper. Semantics of protocol modules composition and interaction. In *5th Int'l Conf. on Coordination Models and Languages*, volume 2315 of *Lecture Notes in Computer Science*, pages 389–404. Springer, April 2002.

[WMS02b] P. T. Wojciechowski, S. Mena, and A. Schiper. Semantics of protocol modules composition and interaction. Technical Report IC-2002/02, School of Computer and Communication Sciences, EPFL, February 2002. A shorter version appeared in Proceedings of Coordination 2002 (The Fifth International Conference on Coordination Models and Languages), April 2002.

[WRS04] P. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: Framework for synchronization augmented microprotocol approach. In *Proc. of Int. Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, US, 2004.

[WS04] M. Wiesmann and A. Schiper. Beyond 1-safety and 2-safety for replicated databases: Group-safety. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT2004)*, Heraklion - Crete - Greece, March 2004.

[WS05] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, April 2005.

[Yos01] T. Yoshida. Message ordering based on the strength of causal relation. In *15th Int'l Conf. on Information Networking (ICOIN)*, pages 915–920, Beppu, Japan, February 2001.

List of publications

Published Parts of this Thesis

Chapter 3

[MCGS03] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia *vs.* Cactus: Comparing protocol composition frameworks. In *Proc. of 22nd IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy, October 2003.

Chapter 4

[UDMK06] 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, Kanazawa, Japan, March 2006.

Chapter 5

[BMN05a] D. Bünzli, S. Mena, and U. Nestmann. Protocol composition frameworks: A header-driven model. In *Proc. of 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA'04)*, Cambridge, MA, USA, July 2005.

[BMN05b] D. Bünzli, S. Mena, and U. Nestmann. Protocol composition frameworks: A header-driven model. Technical Report IC/2005/07, École Polytechnique Fédérale de Lausanne, Switzerland, March 2005. Extended version.

Chapter 7

[MSW03a] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. of ACM/IFIP/USENIX 4th International Middleware*

Conference (Middleware'03), Springer LNCS, Vol 2672. Rio de Janeiro, Brazil, June 2003.

[MSW03b] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. Technical Report IC/2003/01, École Polytechnique Fédérale de Lausanne, Switzerland, January 2003.

Chapter 8

[MS05a] S. Mena, and A. Schiper. A new look at atomic broadcast in the asynchronous crash-recovery model. In *Proc. of 24th IEEE International Symposium on Reliable Distributed Systems (SRDS'05)*, Orlando, FL, USA, October 2005.

[MS05b] S. Mena, and A. Schiper. A new look at atomic broadcast in the asynchronous crash-recovery model. Technical Report IC/2004/101, École Polytechnique Fédérale de Lausanne, Switzerland, December 2004.

Chapter 10

[MBK+05] S. Mena, C. Basile, Z. Kalbarczyk, A. Schiper, and R. K. Iyer. Assessing the crash-failure assumption of group communication protocols. In *Proc. of 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, Chicago, IL, USA, November 2005.

Other Publications in the Context of this Thesis

[BFM+06] D. Bünzli, R. Fuzzati, S. Mena, U. Nestmann, O. Rütti, A. Schiper, and P. T. Wojciechowski. Advances in the design and implementation of group communication middleware. In *Dependable Systems: Software, Computing, Networks*, Springer LNCS, Vol 4028. September 2006.

[EMPS04a] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards flexible finite-state-machine-based protocol composition. In *Proc. of 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA'04)*, Cambridge, MA, USA, August 2004.

[EMPS04b] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards flexible finite-state-machine-based protocol composition. Technical Report IC/2004/63, École Polytechnique Fédérale de Lausanne, Switzerland, July 2004. Extended version.

[WMS02a] P .T. Wojciechowski, S. Mena, and A. Schiper. Semantics of Protocol Module Composition and Interaction. In *Proc. of 5th International Conference on Coordination Models and Languages (Coordination'02)*, Springer LNCS, Vol 2315. York, United Kingdom, April 2002.

[WMS02b] P .T. Wojciechowski, S. Mena, and A. Schiper. Semantics of Protocol Module Composition and Interaction. Technical Report IC/2002/02, École Polytechnique Fédérale de Lausanne, Switzerland, February 2002. Extended version.

Curriculum Vitæ

I was born in Valencia (Spain) at the end of 1975. I attended primary and secondary school in Valencia. From 1985 to 1989, I attended weekly courses at CETISA Study Center (Valencia), where I had my first contact with computer science. In 1993, I graduated from *San José de Calasanz* High School, also in Valencia, and started studying computer science at the Technical University of Valencia. I graduated with a B.Sc. (1996), and a M.Sc. (1999) degrees in Computer Engineering, with distinction. In March 1997, I was selected for a 12-month-long computer science internship at Iberia Spanish Airlines, in Madrid. In July 1999, 12 days after finishing my degree, I started working at Norsistemas (Madrid), nowadays called Soluziona, where I was a junior consultant. I served in this post for six months, until I started my military service, back in Valencia. In February 2000, I was hired at the Information Technology Area of Iniciativas (Valencia), a state-subventioned enterprise devoted to prevention of working environment accidents, where I performed system administration tasks, as well as web development. In October 2000, I moved to Switzerland to follow the Graduate School in Computer Science at the Ecole Polytechnique Fédérale de Lausanne. As part of the Graduate School, I carried out a project on modular group communication in the Distributed Systems Laboratory (LSR) under the supervision of Professor André Schiper. Since October 2001, I have been working at LSR as a research and teaching assistant and a PhD student under the guidance of Professor André Schiper.