Prioritized Atomic Multicast Protocols

Adding prioritization support to atomic multicast protocols

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

Tesis Doctoral

Presentada por:
Emili Miedes De Elías

Dirigida por:
Dr. Francisco Daniel Muñoz Escoí

Mayo de 2012

Contents

List of Figures

List of Tables

Abstract

Group communication has been studied intensively during the last decades from a theoretical and practical point of view. Nowadays, there exists a large number of results related to various topics like group membership, reliable message broadcast, message broadcast with some kind of additional guarantees, distributed agreement, leader election, etc. Thanks to these results, we now have a set of techniques and tools that allow us to design, develop and validate highly diverse complex distributed systems. Replicated systems are a type of systems that can benefit from these techniques and tools. Such systems can be used to build distributed applications more available and fault-tolerant.

One of these topics, the reliable message broadcast with total order guarantees, studies how the different components of a distributed system in general (and a replicated one, in particular) can broadcast messages so that all components receive the same sequence of messages (this is, in the same order). To solve this problem, different algorithms and protocols have been devised. At present, there is a large number of theoretical results and practical tools that solve this problem.

In this thesis, we study an *extension* of this problem. The goal is to study how to add a new *prioritization* guarantee. This guarantee has to allow the components of a distributed application to broadcast messages. The main difference from the *classic* total order algorithms is that the application can label the messages with a *priority* expressing the *importance* or *urgency* of each message. The problem to solve is to ensure that all the components of the distributed system will receive the same sequence of messages and moreover, to ensure that the high-priority messages are received *before* the low-priority ones.

To solve this problem a new *prioritized* total order algorithm could be designed. However, the approach followed in this thesis consists in studying how to modify, from a *theoretical* point of view, some existing algorithms and protocols by *adding* to them the necessary support of prioritized messages. The advantage of this approach is its generic nature, since the results can be applied to existing algorithms and protocols. In the thesis, such a study is performed and later some practical experiments are carried out to ensure that the proposed *theoretical* prioritization techniques are effective and do not impose an excessive overhead in terms of the amount of resources used.

The thesis is completed with a proposal of a mechanism to dynamically switch total order protocols. This mechanism allows an application to switch, in run-time, the total order protocol it is using.

Resumen

La comunicación a grupos ha sido estudiada intensamente durante las últimas décadas desde un punto de vista tanto teórico como práctico. Actualmente se dispone de un gran número de resultados relacionados con diversos temas como la pertenencia a grupos, la difusión fiable de mensajes, la difusión de mensajes con algún tipo de garantía adicional, el acuerdo distribuido, la elección de líder, etc. Gracias a estos resultados, actualmente se dispone de un conjunto de técnicas y herramientas que permiten diseñar, desarrollar y validar sistemas distribuidos complejos de muy diverso tipo. Los sistemas replicados son tipos de sistemas distribuidos que pueden beneficiarse de dichas técnicas y herramientas. Este tipo de sistemas puede usarse para construir aplicaciones distribuidas más disponibles y tolerantes a fallos.

Uno de estos temas, la difusión de mensajes con garantía de orden total, estudia de qué manera es posible hacer que los distintos componentes de un sistema distribuido en general (y uno replicado, en particular) puedan difundir mensajes de manera que todos los componentes reciban la misma secuencia de mensajes (es decir, en el mismo orden). Para resolver este problema, se han ideado distintos algoritmos y protocolos. Actualmente se dispone de una gran cantidad de resultados teóricos y de un buen número de herramientas prácticas que lo resuelven.

En la presente tesis se estudia una *extensión* de este problema. El objetivo es estudiar la forma de añadir una nueva garantía de *priorización*. Esta garantía debe permitir a los componentes de una aplicación distribuida difundir mensajes. La diferencia fundamental respecto a los algoritmos *clásicos* de orden total es que la aplicación puede etiquetar los mensajes con una *prioridad*, que indica la *importancia* o *urgencia* de cada mensaje. El problema a resolver es asegurar la garantía original de que todos los componentes del sistema distribuido van a recibir la misma secuencia de mensajes y además, garantizar que los mensajes más prioritarios van a ser recibidos *antes* que los menos prioritarios.

Para resolver este problema se podría optar por diseñar algún nuevo algoritmo de orden total *priorizado*. Sin embargo, la aproximación seguida en la tesis consiste en estudiar de forma *teórica* la forma de modificar algunos algoritmos y protocolos de orden total existentes y *añadirles* el soporte de mensajes priorizados. La ventaja de esta aproximación es su genericidad, ya que permite que los resultados puedan ser aplicados a algoritmos y protocolos de orden total existentes. En la tesis se realiza dicho estudio y posteriormente se realizan algunos experimentos prácticos para garantizar que realmente las técnicas de priorización *teóricas* propuestas son efectivas y que no imponen un gran sobrecoste en términos de la cantidad de recursos utilizados.

La tesis se completa con la propuesta de un mecanismo de intercambio dinámico de protocolos de orden total. Este mecanismo permite a las aplicaciones que utilizan un protocolo de orden total, cambiarlo en tiempo de ejecución.

Resum

La comunicació a grups ha sigut estudiada intensament durant les darreres décades des d'un punt de vista tant teòric com pràctic. Actualment es disposa d'un gran nombre de resultats relacionats amb diferents temes com la pertinença a grups, la difusió fiable de missatges, la difusió de missatges amb algun tipus de garantia addicional, l'acord distribuït, l'elecció de líder, etc. Gràcies a aquests resultats, actualment es disposa d'un conjunt de tècniques i eines que permeten dissenyar, desenvolupar i validar sistemes distribuïts complexes de múltiples tipus. Els sistemes replicats són sistemes distribuïts que poden beneficiar-se d'aquestes tècniques i eines. Aquest tipus de sistemes pot utilitzar-se per a construir aplicacions distribuïdes altament disponibles i tolerants a les fallades.

Un d'aquests temes, la difusió de missatges amb garantia d'ordre total, estudia de quina manera es pot aconseguir que els diferents components d'un sistema distribuït en general (i d'un sistema replicat, en particular) puguen difondre missatges de manera que tots els components reben la mateixa seqüència de missatges (és a dir, en el mateix ordre). Per a resoldre aquest problema s'han dissenyat diferents algorismes i protocols. Actualment es disposa d'una gran quantitat de resultats teòrics i d'un bon nombre d'eines pràctiques que el resolen.

En aquesta tesi s'estudia una *extensió* d'aquest problema. L'objectiu és estudiar la manera d'afegir una nova garantia de *priorització*. Aquesta garantia ha de permetre que els components d'una aplicació distribuïda difonguen missatges. La diferència fonamental front als algorismes *clàssics* d'ordre total és que l'aplicació pot etiquetar els missatges amb una *prioritat* que indica la *importància* o *urgència* de cada missatge. El problema a resoldre es garantir que tots els components del sistema distribuït reben la mateixa seqüència de missatges i, a més, que els missatges més prioritaris siguen rebuts *abans* que els menys prioritaris.

Per a resoldre aquest problema es podria optar per dissenyar algun nou algorisme d'ordre total *prioritzat*. No obstant això, l'aproximació que hem seguit consisteix en estudiar *teòricament* com modificar alguns algorismes i protocols d'ordre total existents i *afegir-los* el suport de priorització de missatges. L'avantatge d'aquesta aproximació és la seua genericitat, ja que permet que els resultats puguen ser aplicats a algorismes i protocols d'ordre total existents. En la tesi es fa aquest estudi i posteriorment es realitzen alguns experiments pràctics per a demostrar que realment les tècniques de priorització *teòriques* són efectives i que no imposen un sobrecost excessiu en termes de quantitat de recursos utilitzats.

La tesi es completa amb la proposta d'un mecanisme d'intercanvi dinàmic de protocols d'ordre total. Aquest mecanisme permet que les aplicacions puguen canviar de protocol en temps d'execució.

Part I

Introduction

Chapter 1

Introduction

1.1 Group Communication

The design of a distributed system is a complex task because of the large number of issues and problems that must be considered and solved. Some of them are inherent to the distributed nature of the system.

One of the goals to cover is to reach a high level of *transparency*. Informally, the system is expected to behave, at least from the point of view of the final user, as a *stand-alone* system, but at the same time, it may offer a number of advantages, like a certain degree of *fault tolerance*, higher *availability* of the application, a better performance or a better use of the resources.

In this context, there exists a family of problems globally known as *consensus problems* [44] based on a common idea: the need to reach an agreement among the different nodes of the system (for instance, different processes running in different geographically sparse machines). There are a number of particular well-known cases of this problem that have been studied during the last decades. One of these particular cases is the problem of *distributed agreement* that consists in reaching some agreement among the different nodes of the system on the value a variable *shared* among all of them. Another particular case is the *leader election* [45] problem. This problem consists in electing a member as a *leader* among all the members of a group.

Another well-known case is the problem of *atomic multicast* [49], also known as *total order broadcast*. Informally, the members of a group broadcast messages to all the nodes of the system and the problem to solve consists in getting all the messages *totally ordered* so each node receives the same sequence of messages.

The present thesis is about *total order protocols with prioritization support*, a variation of the classic problem of total order. According to the conventional definitions of total order, all the messages broadcast by the members of the system have the same *importance* or *priority.* In this thesis, we study how to modify the existing *classic* total order broadcast algorithms and protocols so they allow the applications to set the *priority* of each message and later consider those priorities when deciding the *global* sequence of ordered messages.

The consensus problems pointed out above and in general, many other problems related with the design and development of distributed systems might be straightforward from a theoretical point of view when a comfortable system model is assumed. In fact, the solution of this kind of problems in *ideal* settings is quite simple. Nevertheless, there are a number of issues related with the implementation and, especially, with the deployment of these solutions that may cause important complications. These complications are so large that usually their solution can't be directly addressed in the implementation phase. Instead, they must be considered previously, in design time.

Some of those complications are related to the *asynchronous* nature of the computers and communication networks that can be used nowadays to solve these problems. They appear when the systems are *moved* from and *ideal* to a more *real* setting. In particular, these complications appear when the design of a distributed system considers the different types of *asynchrony* that are present in current computers and communication networks. We can refer to two types of well-known types of asynchrony. The first type is related to the computer processors. If no precautions are taken, the processors that run the different nodes of a distributed system make progress at different speeds and in general, it is not easy to *synchronize* their progress.

On the other hand, there is also an important degree of asynchrony in the communication networks and, in particular, in the time needed to send a data packet from a sender to a receiver. A number of intermediate elements take part in the packet sending, from physical low-level devices (network adapters, modems, routers, bridges, etc.) to software components piled up in protocol stacks (like TCP/IP) and each one is adding a small delay that are noticeable as a whole by both sender and receiver as *one-way* and *round-trip* latencies. The variability of these latencies is so large that, in general, it is not possible to precisely know a bound of the time needed by a packet to reach its destination or even if such a bound exists.

Another issue the designers of distributed systems must deal with is the possibility of *failures* of different types, that may be caused by different reasons, like physical failures (for instance, due to the regular use and wear of the physical parts of the system), software bugs, security threats, human errors and natural disasters.

For the system to be useful, these issues must be foreseen and solved, by means of a number of solutions, in both design time and development time. These tasks require a great deal of knowledge about different areas of computing (computing theory, algorithm design, software architecture and design, communication networks, operating systems, programming languages, etc.)

Due to all these issues and their *inherent complexity*, the design and implementation of a distributed system becomes complicated. A number of techniques can be applied in order to reduce this complexity or even just make these tasks affordable. One of the *always winning* strategies consists in adopting a *modular* methodology, by dividing the system in smaller parts, clearly *isolating* those problems that are especially difficult to solve. A second step consists in getting solutions to these problems, for instance developed by third parties, choosing the more suitable to each problem, depending on the available resources, the current limitations and in general, any other form of *context* that may be considered appropriate. Finally, those solutions must be integrated into the system under development as *building blocks*, by means of well-defined interfaces.

This modular approach has a number of advantages. First, it simplifies the design and implementation of the applications. For instance, the designer of a distributed application does not need to be an expert in a large large number of *low level* topics like networking or advanced administration of operating systems. Instead, the designer relies on existing solutions that solve the problems in an effective way. Moreover, it implies a significant saving, in terms of resources like time and budget, because the design and development efforts that would be required are no longer needed.

In this thesis, we defend the use of well-known architectures and design patterns, based on well-defined interfaces that favor the integration of existing solutions and also the modular development of new solutions that can be integrated in other systems.

1.2 Goals and Structure of the Thesis

As pointed out in the previous section, the *classic* definition of total order assumes that all the messages broadcast by the nodes of the distributed system have the same importance or *priority.* But a distributed application running in different nodes may need to broadcast messages with different *priorities*[1], so all the messages broadcast by the nodes are totally ordered, according to the priorities set in the messages. Informally, the applications expect that the messages with a *higher* priority will be delivered *before* the messages with a *lower* priority.

Nevertheless, the *classic* total order protocols have not been designed to consider prioritized messages. Instead, they consider that all the messages have the same priority. Thus, the application designers that need to use such a service in their applications, will need to choose one of the following alternatives.

A first alternative is to develop their own *ad-hoc* solutions, i.e., solutions *specifically tailored* to the application, as any other part or *module* of the application, with the complication that it entails. Another alternative consists in designing

[1]In Chapter 4 we show an example and use it as a *case study.*

from scratch and implementing a new total order protocol, including in its design the mechanisms to consider the priorities of the messages as a criterion to decide the total order of the messages broadcast by the nodes. A third alternative consists in choosing an existing implementation of a total order protocol and modify it to consider the priorities of the messages. These alternatives require a particular knowledge and also a considerable effort.

In this thesis we try to explore a different alternative, which we consider wider and more general than the previous ones. Instead of modifying any existing implementation of a total order protocol, our alternative consists in studying, from a *theoretical* point of view, how can we modify the classic total order *algorithms and protocols* so they consider the priorities of the messages. The result of this study has to provide us with a number of *techniques* to be applied to existing implementations of total order protocols in order to get their corresponding *prioritized* versions that can then be provided to the application designers, to be used as *building blocks* to develop their applications.

Besides designing and developing prioritized protocols, this thesis also proposes a meta-protocol able to support multiple plugged-in total order broadcast protocols and dynamically exchange the one being used. Both techniques (prioritization and dynamic protocol exchange) are samples of adaptive strategies for current dynamic distributed systems. Prioritization gets its best results with high message sending rates, and current distributed systems are focused on improving their scalability levels, generating such sending rates. So, prioritization could be a complementary technique for reducing the transaction abort rates in replicated relational databases (as explained in Chapter 4). On the other hand, dynamic protocol exchange improves the adaptability of the system to varying workloads, selecting the best broadcast protocol for each workload range. As a result, this thesis presents some contributions to improve the adaptability and dynamicity of the total order broadcast mechanisms being used in a reliable distributed system.

Other results exist in this same area that complement our contributions, although they have not been analyzed in subsequent chapters, which are only centered in the prioritization and dynamic exchange problems for total order broadcast protocols. A sample of these related works is the usage of system interconnection [17] approaches, enhancing the scalability and adaptability of the message broadcast mechanisms. Unfortunately, broadcast interconnection algorithms can not be used for total order delivery, as proven in [9], but only for causal or FIFO delivery. This is the reason for not studying them in the rest of this thesis.

The thesis is structured in five main chapters. In Chapter 2, we review some basic *Group Communication* concepts. In Chapter 3 we propose a number of techniques that can be used to modify the different existing total order algorithms and protocols. In Chapter 4 we apply these techniques to specific implementations of different *classic* total order protocols and provide an experimental evaluation to prove their effectiveness. In Chapter 5 we provide a second experimental evaluation to prove that the mechanisms that implement those *prioritization techniques* do not impose a significant performance overhead to the original total order protocols to which they are applied. In Chapter 6 we provide a mechanism to change dynamically (i.e. in run-time) the total order protocol (prioritized or *regular*) used by an application, to increase its *adaptability* to changing environments.

1.3 Group Communication: Theoretical Issues

1.3.1 Fundamental Results

During the last three decades, Group Communication has been studied from both a theoretical and a practical point of view and a great number of results have been produced. In this section, we present a selection of the most relevant in the context of this thesis.

In [61], Lamport presents his definition of *causal relation* and uses it to define a *partial order* that allows to order the events of a system in a *logical* manner that is independent from the *physical* order in which they happened. This work is essential for many other subsequent papers by many other authors. For instance, it is the basis of [53], where Parker shows his definition of *vector clock*, which in its turn is the basis of several causal broadcast protocols.

In [26, 27] (and some previous related papers), Birman et al. show the concept of *Virtual Synchrony*, based on a number of *fundamental abstractions* like some *atomic multicast* primitives (ABCAST, CBCAST and GBCAST) and some concepts and abstractions related with *groups* (as the *group* concept itself and some *actions* like *group join*, *group leave* or the monitoring of changes in the composition of groups). Informally, the *Virtual Synchrony* property says that given two nodes p and q that belong to *group view* V, if both switch to *group view* W then both nodes deliver in V the same set of messages, before switching to W[2].

In [48], Hadzilacos and Toueg show a specification of different properties of different message broadcast mechanisms, specifically, reliable message broadcast and ordered message broadcast mechanisms, including different ordering semantics (FIFO, causal and total order). They give their definition of some properties like *Validity*, *Integrity*, *Agreement*, *Uniformity*, *Termination* and several properties to define different types of ordered message broadcast protocols. This specification, unlike others, can be considered *static*, in the sense that it assumes static systems, i.e. systems whose composition does not change over time.

[2]Despite being a fundamental *Group Communication* concept, it will not be taken into too much account in the stages of this thesis since we focus on the total order protocols included in a regular *Group Communication System* and assume that this includes a membership service that offers the *Virtual Synchrony* property. Nevertheless, when we talk about the correction of the proposed *protocol switching mechanism* we will pay some attention to the relationship it has with the available membership service (see Sections 6.3.3 and 6.4).

In [13], Babaoglu et al. present the concept of *View Synchrony* as an extension of the concept of *Virtual Synchrony* for partitionable systems.

In [43], Fekete et al. provide a formal specification of the problem of group communication in partitionable systems.

In [14, 15], Babaoglu et al. extend the work in [26] to the case of partitionable systems, by extending the concept of *Extended Virtual Synchrony* previously presented by Moser in [81]. They provide another property specification that can be used in partitionable systems whose components may evolve independently over time.

In [33], Chockler et al. present an extensive survey of many total order multicast and broadcast algorithms and protocols and classify them according to their ordering and fault-tolerance mechanisms.

Finally, it is worth mentioning the relation between group communication and replication, which has been extensively studied. For instance, in [47, 58, 85, 104, 102, 12, 59] there are a number of proposals on how to use group communication techniques and abstractions and services to design replicated systems.

1.3.2 Implementation Level

In [52], Chang and Maxembuck present one of the first atomic broadcast protocols, based on the setup of a *logical ring* with the processes of the system and the circulation of a special message. This idea was exploited later in many atomic broadcast algorithms and protocols.

In [54], Kaashoek and Tanenbaum present an introduction to the Amoeba distributed operating system, which includes one of the first implementations of sequencer-based total order broadcast protocols (see Section 2.1).

Another of the well-known *classical* group communication systems is the Isis Toolkit [25, 91], which is a monolithic system (i. e. all the available services like the membership service, the broadcast primitives, etc. are tightly coupled in a single *module* and are thus inter-dependent). The Horus [99, 98] and Ensemble [97] systems are evolutions of Isis which adopt a modular architecture composed by a configurable *stack of layers*. In these systems, and in many other that later adopted the same kind of architecture, any layer is based on the services offered by the *lower* layers and in its turn, offers a number of services to the *upper* layers.

In [82, 11], the Totem group communication system is presented. It is another monolithic system which also has an architecture based on a token-ring, that can be extended with a *multiple ring* architecture to be used in a highly scalable system. Unlike some other toolkits, Totem is able to deal with partitionable systems.

In [42], the Transis toolkit is presented. As Totem, Transis offers group communication services to software applications that run in partitionable systems.

In case a partition happens and the set of processes is divided in two or more components, Transis allows all of them to make progress and later *re-merge*.

Among the most current implementations, we can cite a few. The Spread Toolkit [10, 1] is an open source group communication system written in C which inherits a number of characteristics of Totem and Transis, including its ability to deal with partitionable systems and its token-ring architecture. The JGroups toolkit [2], formerly known as JavaGroups is another open source group communication system written in Java that has a fully configurable multilayer architecture partially inspired in the Horus and Ensemble layer stacks. The Appia toolkit [78, 3] is another open source group communication system also written in Java which has a configurable multilayer architecture. In [18], a comparison among Spread, JGroups and Appia can be found.

1.4 Publications

According to the requirements of the internal regulations of the PhD Program of the Universitat Politècnica de València, in this section a selection of the publications of the author of this thesis is presented.

1. Emili Miedes, Mari-Carmen Bañuls, and Pablo Galdámez. Comparando Protocolos mediante JavaGroups. In *I Congreso Internacional de Cómputo Paralelo, Distribuido y Aplicaciones*, Linares, México, September 2003.

2. Emili Miedes, Mari-Carmen Bañuls, and Pablo Galdámez. An Adaptive Group Communication System. In *1st Polish and International PD Forum-Conference on Computer Science*, Bronislawow, Lodz, Poland, April 2005. ISBN 83-60434-25-5.

3. Emili Miedes, Mari-Carmen Bañuls, and Pablo Galdámez. Group Communication Protocol Replacement for High Availability and Adaptiveness. In *Advanced Distributed Systems: 6th International School and Symposium (ISSADS)*, Guadalajara, México, January 2006.

4. Juan Carlos García, Mari-Carmen Bañuls, Pablo Galdámez, and Emili Miedes. Membership Estimation Service for High Availability Support in Ad Hoc Networks. In *Advanced Distributed Systems: 6th International School and Symposium (ISSADS)*, Guadalajara, México, January 2006.

5. Stefan Beyer, Francesc D. Muñoz-Escoí, Pablo Galdámez, and Emili Miedes. DeDiSys Lite: An Environment for Evaluating Replication Protocols in Partitionable Distributed Object Systems. In *XIV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, San Sebastián, Spain, June 2006. ISBN 84-689-9292-5.

6. Juan Carlos García, Mari-Carmen Bañuls, Pablo Galdámez, and Emili Miedes. A Study of the Trade-Off Between Power Consumption and Membership Estimation in Ad Hoc Networks. In *In XIV Jornadas de Concurrencia y Sistemas Distribuidos (JCSD)*, pages 271–283, San Sebastián, Spain, June 2006. ISBN 84-689-9292-5.

7. Emili Miedes and Francesc D. Muñoz-Escoí. Managing Priorities in Atomic Multicast Protocols. In *International Conference on Availability, Reliability and Security (ARES)*, Barcelona, Spain, March 2008. ISBN 0-7695-3102-4.

8. Luís Rodrigues, Nuno Carvalho, and Emili Miedes. Supporting Linearizable Semantics in Replicated Databases. In *7th IEEE International Symposium on Network Computing and Applications (NCA08)*, Cambridge, MA, USA, July 2008. ISBN 978-0-7695-3192-2.

9. Emili Miedes, Francesc D. Muñoz-Escoí and Hendrik Decker. Reducing Transaction Abort Rates with Prioritized Atomic Multicast Protocols. In *14th International European Conference on Parallel and Distributed Computing (Euro-Par)*, Las Palmas de Gran Canaria, Spain, August 2008. Lecture Notes In Computer Science (LNCS), vol. 5168, pages 394–403. Springer-Verlag, Heilderberg (Germany). ISBN 978-3-540-85450-0.

10. Ken Mayes, Juan Carlos Garcia Ortiz, Emili Miedes, and Stefan Beyer. Reliable Group Communication for Dynamic and Resource-Constrained Environments. In *International Workshop on Database and Expert Systems Applications (DEXA)*, pages 14–18. IEEE Computer Society, Linz, Austria, September 2009. ISBN 978-0-7695-3763-4.

11. Emili Miedes and Francesc D. Muñoz-Escoí. On the Cost of Prioritized Atomic Multicast Protocols. In *11th International Symposium on Distributed Objects, Middleware and Applications (DOA)*, Vilamoura, Portugal, November 2009. Lecture Notes In Computer Science (LNCS), vol. 5870, pages 585—599. Springer-Verlag, Heilderberg (Germany). ISBN 978-3-642-05147-0.

12. Emili Miedes and Francesc D. Muñoz-Escoí. Dynamic Switching of Total-Order Broadcast Protocols. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, Nevada, USA, July 2010. CSREA Press. ISBN 1-60132-158-9.

From the previous listing, references 2, 3, 7, 9, 11 and 12 are directly related to this thesis. This work is original from the author, it does not appear as part of any other Ph.D. thesis and none of the papers in this group of publications will support any future thesis written by any other of the paper co-authors. The remaining references (1, 4, 5, 6, 8 and 10) are not directly related to this thesis.

Chapter 2

Preliminaries

This chapter provides a number of preliminaries. In Section 2.1 we review an existing classification of total order protocols. In Section 2.2 we present the system model we assume for the rest of the chapters. In Section 2.3 we propose a formal definition of ***Prioritized Total Order***. Finally, in Section 2.4 a number of issues related to the prioritization of total order protocols is presented.

2.1 Reviewing Total Order Protocols

In [39], a survey of total order protocols is given. Such work classifies total order protocols in five different classes: ***fixed sequencer***, ***moving sequencer***, ***privilege-based***, ***communication history*** and ***destinations agreement*** protocols.

The following sections include a review of that classification and a proposal of how to enable the given classes of protocols to totally order messages in a priority-based manner.

2.1.1 Fixed Sequencer Protocols

In a ***fixed sequencer*** protocol, a single process is in charge of ordering the messages. If no process fails, this special process is fixed. There are three different versions of the ***fixed sequencer*** basic protocol, as identified in [39], which extends the classification in [54]: the Unicast-Broadcast (UB) protocol, the Broadcast-Broadcast (BB) protocol and the Unicast-Unicast-Broadcast (UUB) protocol.

In the UB version, when a sender wants to broadcast a message to a set of processes, it first sends (unicasts) it to the sequencer. Other processes can concurrently send their own messages to the sequencer and it receives them in some order. When the sequencer receives a message, tags the message with a global sequence number and broadcasts it. All the processes deliver the messages in the order set by the sequence number of each message.

In the BB version, a sender broadcasts the message to all the processes. The sequencer receives different messages broadcast from different senders, in some order. Every message is tagged with the sender identifier and a sequence number local to its sender. When the sequencer receives a new message, it assigns the message a global sequence number and broadcasts a special message containing the original sender identifier and local sequence number, and the global sequence number. All the processes deliver the messages according to the global sequence numbers sent by the sequencer.

In the UUB version, a sender sends a request message to the sequencer, which answers with a sequence number. The sender then tags the message to send with that sequence number and broadcasts it. All the processes deliver the messages in the right order, depending on their sequence number.

In [54], a comparison of the UB and the BB protocols (their original names are PB and BB) can be found.

2.1.2 Moving Sequencer

In a ***moving sequencer*** protocol, sequencing is performed by a single process, as in a ***fixed sequencer*** protocol, but in this case the sequencer is not a fixed process. The sequencer role is transferred from one process to another, among a set of processes that can be the whole set of processes in the system or just a subset.

In several implementations [39], all the processes in the system form a logical ring and the sequencer role is transferred along the ring by means of some kind of token message.

The actual method to order the messages can be any of the methods used by a fixed sequencer (UB, BB or UUB), but some details must be considered when merging the UB, BB or UUB ordering methods with the ring-based method of transferring the sequencer role.

For instance, the combination of the UB method and a logical ring presents an important disadvantage. Whenever the sequencer role is transferred to another process, the current sequencer must send to the new sequencer the whole set of incoming messages that have not been sequenced yet. If there are too many pending messages, this transfer may involve a significant bandwidth cost and impose a significant delay in the regular message ordering.

2.1.3 Privilege-Based Protocols

In a ***privilege-based*** protocol, processes can only send messages when they are allowed to do it. If just one process is allowed to send messages at every moment, then the total order can easily be set using just a global sequence number.

Common implementations (e.g. [82, 10]) use a logical ring composed of all the processes in the system. A special message or token is sent along the ring. Only the owner of the token is allowed to send messages. A process that wants to send some messages must wait until it receives the token.

The token contains a global sequence number. Before sending a message, the current token holder tags it with the current sequence number in the ring and then increments it, so the next message, sent by the current sender or by another one, gets the next sequence number. Once the sender has sent its messages, forwards the token to the next process in the ring. Processes deliver the message according to the sequence number set in it.

This protocol is restricted to closed groups (i. e., all the senders know each other), which fits the system model. Some kind of static configuration or membership service is needed.

Moreover, some kind of flow control is needed, to ensure that processes do not send too many messages in its turn and do not keep the token for too much time.

2.1.4 Communication History Protocols

In a ***communication history*** protocol, processes use historical information about message sending, reception and delivery to totally order messages.

In [39], two different types of ***communication history*** protocols are identified: ***causal history*** protocols and ***deterministic merge*** protocols.

Causal History Protocols

The class of ***causal history*** protocols is based on the total order mechanism proposed in [61]. The idea is to causally order messages tagged with Lamport clocks, and extend this causal order into a total order. Although causal order imposes a partial order on the messages that are logically dependent, it is not enough to totally order concurrent messages (informally, messages that are causally independent). These are ordered using the identifier of the message sender.

Deterministic Merge Protocols

In a ***deterministic merge*** protocol, messages are also broadcast with some kind of timestamp, but unlike ***causal history*** protocols, these timestamps do not reflect causal relations among messages. In practice, this timestamp can even be a local sequence number. On the other hand, receivers use some local deterministic mechanism to totally order the messages.

In [39], several ***deterministic merge*** protocols are presented ([34, 57, 7, 36, 37, 21]), although most of them impose significant constraints on the system they may be used in (there are protocols for synchronous systems, protocols that depend on physical clocks or even on redundant reliable channels) so they will not be considered.

A particular deterministic merge protocol. In [39], a couple of ***deterministic merge*** protocols ([20, 19]) based on a round-robin algorithm are cited. As they are good examples of the ***deterministic merge*** class of protocols, here it is presented how they can be adapted to consider properties when deciding the total order of the messages.

According to those algorithms, all the processes send a constant flow of messages (that may be *dummy* messages if the user application does not send enough messages), that are tagged with a local sequence number. In each round of the protocol, every process waits for and delivers a message from the first process, then another one from the second process and so on. Once a process has received and delivered a message from all the processes in the system, another round is started. As all the processes are deterministically ordered (by its process identifier) and all the messages sent by a process are also deterministically ordered (by its local sequence number), total order can be easily guaranteed.

This protocol works in a static scenario, in which the set of processes is fixed and their identities are well-known. For this protocol to work in a more dynamic environment, some additional group membership support is needed, to allow a node to know about processes that join the group, leave the group or fail.

On the other hand, this protocol only makes an efficient use of the network when all the nodes have a constant flow of messages to send. If not, dummy messages need to be sent, thus wasting network bandwidth.

2.1.5 Destinations Agreement Protocols

In a ***destinations agreement*** protocol, some kind of agreement protocol is run to decide the order of one or more messages.

In [39], three subclasses of ***destinations agreement*** protocols are identified, according to the type of agreement performed: (1) agreement on the order (sequence number) of a single message, (2) agreement on the order (sequence numbers) of a set of messages and (3) agreement on the acceptance of an order (sequence numbers) of a set of messages, proposed by one of the processes.

Destinations Agreement on the Order of a Single Message (Subclass 1)

An example of such a protocol is an algorithm originally proposed by Skeen and later modified and formalized in [27]. In this protocol, message ordering is performed in three phases (including a broadcast phase). First of all, a process broadcasts a message. Receivers propose a sequence number, and also send their own process identifiers. The sender collects all the proposed sequence numbers and then deterministically decides the final sequence number. Process identifiers are used to break ties, in case that two or more processes propose the same local identifier.

Destinations Agreement on the Order of a Set of Messages (Subclass 2)

An example of this class of protocols is the protocol proposed in [30]. This protocol tries to run a sequence of consensus runs. Each of them is used to agree on the order of a set of messages. All the messages whose order is decided in a consensus run are delivered before any message whose order is decided in the next run. In a particular run, total order is decided in a deterministic manner (for instance, according to the order imposed by the message identifiers, using process identifiers to break ties).

In such a protocol, besides the consensus protocol itself, a reliable message transport is needed. A mechanism to decide the scope of each consensus run (i. e. which messages are ordered by the current consensus run and which ones are ordered by the next one) is also needed.

Destinations Agreement to Accept a Suggested Order (Subclass 3)

An example of this class of protocols is the protocol described in [65]. According to this protocol, when a process broadcasts a message, every process locally saves it in a list of incoming messages. Periodically, some *starter* process decides to start a new consensus run to order a set of messages. To take part in a consensus run, a process sends to the starter process some information about the messages contained in its incoming list. The started process waits for, at least, a majority of responses and then decides a total order for the set of messages. This proposed order is sent to all the processes, which vote to accept or reject it. If the starter process receives a majority of positive votes, the order is accepted and applied by all the processes.

2.2 System Model

The system considered is composed of a set $S = \{node_1, node_2, ..., node_n\}$ of physical nodes. In each node, a process is run. The set of processes in the system is $P = \{p_1, p_2, ..., p_n\}$.

Processes communicate through message passing by means of a *fair lossy channel*. Informally, a fair lossy channel is a channel that is subject to the loss of messages, due to network issues like node disconnections or network partitions, process failures or other reasons. However, a fair lossy channel does not lose all the messages, does not produce new spurious messages, does not duplicate messages and does not change the contents of the messages.

Each node has a multilayer structure, as shown in Figure 2.1. The user level is represented by a distributed client application that uses the services offered by a group communication system (GCS), that is composed of one or more group communication protocols (GCP). Each GCP provides some guarantees like reliable message transport or atomic multicast. The GCS sends to and receives messages from the network and delivers them to the client application according to the guaranties provided by the GCPs.

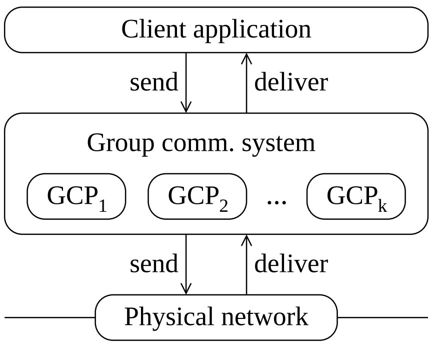


Figure 2.1: Node architecture

Only closed groups are considered. A closed group is a group in which every message sender is also a destination, so no external processes are allowed to multicast messages. If an external process needs to multicast a message to the group, it simply forwards the message to a group member that actually multicasts such message.

The system is partially synchronous [41]. Although several definitions exist on partial synchrony, it is considered that on the one hand, processes run on different physical nodes and the drift between two different processors is not known. On the other hand, the time needed to transmit a message from one node to another is bounded but the bound is not known. In practice, the system does not need more synchrony than that offered by a conventional network which offers a reasonably bounded message delivery time.

Processes can fail due to several reasons (for instance, hardware failures, software bugs or human misoperation). Processes are also subject to network failures that keep them from sending or receiving messages. Network partitions may also occur. Nevertheless, since this work focuses on the comparison of prioritization techniques, these issues will not be addressed here. An implementation of these techniques may rely on some mechanisms like group membership services and fault-tolerance protocols to take care of them.

2.3 Prioritization Properties

Ideally, a priority-based total order broadcast protocol should offer well defined properties regarding the order in which high-priority messages are delivered respect to low-priority messages. Informally, such a property should guarantee that high-priority messages are delivered *first* and low-priority messages can then be delivered.

In this section, the definition of the properties offered by a prioritized total order protocol is formalized.

2.3.1 Previous definitions

As stated in Section 2.2, the system is composed of a set $S = \{node_1, node_2, ..., node_n\}$ of physical nodes. In each node, a process is run and the set of processes in the system is $P = \{p_1, p_2, ..., p_n\}$. A client application is run in each process. The application uses the services offered by a GCS. In a given process, the client application broadcasts a possibly infinite sequence of messages. Thus $M = \{m_1, m_2, \ldots\}$ is the set of messages in the system.

Events. As shown in Figure 2.1, in a process, the client application interacts with the GCS by means of some events. A $send(p, m)$ event is triggered when process p broadcasts a message m through its GCS, where $p \in P$ and $m \in M$. A $deliver(p, m)$ event is triggered when the GCS delivers a message m to process p, where $p \in P$ and $m \in M$.

Traces. A trace is a finite or infinite sequence of events. For instance $T = t_1, t_2, t_3, \ldots$ is a trace, in which event t_1 happens before t_2 which in turn happens before t_3, etc. As in [33], it is assumed that given events t_i and t_j, then $i = j \Leftrightarrow t_i = t_j$, this is, two different events cannot happen at the same time.

Functions. The following definitions are assumed:

- $sender(m) = p$ iff process p sends message m.
- $sender\_index(m) = i$ iff $sender(m) = p_i$.
- $prio(m)$ is the priority of message m.
- $prio(m) > prio(m')$ means that message m has more priority than m'.

It is also assumed the definition of a function *seq* used to know the relative order of two given messages. In particular, $seq(m, m') > 0$ means that message m is delivered prior to m' to the application. Conversely, $seq(m, m') < 0$ means that message m' is delivered prior to m to the application. Moreover, $seq(m, m') \neq 0, \forall m, m' \in M : m \neq m'$

2.3.2 Properties

A prioritized total order broadcast protocol guarantees the following properties: ***Validity***, ***Agreement***, ***Integrity*** and ***Prioritized Total Order***.

P1 Validity. The ***Validity*** property ensures that if a process p sends a message m, then p eventually delivers m (by means of the *deliver* event). More formally:

$t_i = send(p, m) \Rightarrow \exists j : j > i \wedge t_j = deliver(p, m)$

This property is similar to the *Validity* property specified in [49].

P2 Agreement. The ***Agreement*** property ensures that if a process p delivers a message m, then all processes eventually deliver m. More formally:

$t_i = deliver(p, m) \Rightarrow \forall q \in P\ \exists j : t_j = deliver(q, m)$

This property is similar to the *Agreement* property specified in [49]. Moreover, the *union* of the ***validity*** and ***agreement*** properties is equivalent to the *Multicast Liveness* property specified in [33].

P3 Integrity. The ***Integrity*** property ensures that a) a message is delivered by a node at most once and b) it is delivered only if it was sent by its sender (this is, messages are not duplicated and there are no spurious messages). More formally:

a) $t_i = deliver(p, m) \wedge t_j = deliver(p, m) \Rightarrow i = j$
b) $t_i = deliver(p, m) \Rightarrow \exists q\ \exists j\ (j < i \wedge t_j = send(q, m))$

These ***integrity*** a) and b) parts are similar to the *No Duplication* and *Delivery Integrity* properties specified in [33], respectively.

P4 Prioritized Total Order. The ***Prioritized Total Order*** property expresses that if two processes deliver two messages, they deliver them in the same order. More formally:

$seq(m, m') > 0 \Rightarrow t_i = deliver(p, m) \wedge t_j = deliver(p, m') \wedge i < j, \; \forall p \in P, \; \forall m, m' \in Q \subseteq M$

This property is similar to the *Total Order* property in [49]. It is also similar to several definitions of total order in [33] once conveniently tailored to ignore view management details (*Strong Total Order*, *Weak Total Order* and *Reliable Total Order*).

The definition of the *seq* function can be refined to fit a specific class of total order protocols, thus yielding a refined specification of the ***Prioritized Total Order*** property.

Definition of *seq* for *fixed sequencer* protocols.

$$seq(m, m') = \begin{cases} prio(m) - prio(m'), & \text{if } prio(m) \neq prio(m') \\ det_{fseq}(m, m'), & \text{if } prio(m) = prio(m') \end{cases}$$

$\forall p \in P, \; \forall m, m' \in Q$, where

a) det_{fseq} is a deterministic function used to order two messages that have the same priority (for instance, according to the identifiers of their senders, the local sequence numbers of the messages, etc.)

b) $Q = F$ is the set of messages sent to the sequencer process that have not been sequenced yet.

Definition of *seq* for *privilege-based* protocols.

$$seq(m, m') = \begin{cases} prio(m) - prio(m'), & \text{if } prio(m) \neq prio(m') \\ det_{pv}(m, m'), & \text{if } prio(m) = prio(m') \end{cases}$$

$\forall p \in P, \; \forall m, m' \in Q$, where

a) det_{pv} is a deterministic function used to order two messages that have the same priority (for instance, according to the local sequence numbers of the messages, etc.)

b) $Q = G_p$ is the set of messages sent by process p that have not been sequenced yet.

Definition of *seq* for *causal history* protocols.

$$seq(m, m') = \begin{cases} 1, & \text{if } m \to m' \\ -1, & \text{if } m' \to m \\ prio(m) - prio(m'), & \text{if } m \parallel m' \wedge prio(m) \neq prio(m') \\ det_{ch}(m, m'), & \text{if } m \parallel m' \wedge prio(m) = prio(m') \end{cases}$$

$\forall p \in P,\ \forall m, m' \in Q$, where

a) $m \to m'$ means that message m is causally precedent to m' and $m \parallel m'$ means that message m is concurrent to m'

b) det_{ch} is a deterministic function used to order two messages that are concurrent and have the same priority (for instance, according to the identifiers of their senders, the local sequence numbers of the messages, etc.)

c) $Q = H_p$ is the set of messages received by process p that have not been delivered yet.

2.4 Problems

Priority-based ordering usually undergoes two common problems: *starvation* ([84]) and *priority inversion* ([94, 16, 31, 101, 100]).

Starvation happens when the delivery (or even the sending) of a low priority message is delayed for a long period of time due to, for instance, a flow of high priority messages. For instance, consider the following scenario. A process sends a constant flow of high priority messages. Another process sends a constant flow of low priority messages. Some priority-based total ordering protocol is ran. The protocol decides that high priority messages must always be delivered prior to any available low priority message. As there is a constant flow of high priority messages, low priority messages never get a chance to be ordered. If no special care is taken in such a situation, low priority messages are never delivered.

However, due to the nature of the applications that may be run over such a priority-based total order broadcast protocol, the starvation problem is not addressed in this work. Several solutions can be applied to at least minimize the impact of message starvation. For instance, in [83] a time division technique is used to avoid message starvation, without introducing too much *priority inversion*, which is the other common issue to care about.

Moreover, dynamic priority scheduling techniques [86] used by operating system process schedulers may be useful to prioritize too old low priority messages. A common solution consists in dynamically increase the priority of a message if it has been waiting too much time in the sending queue.

Priority inversion happens when high priority messages are forced to wait until some lower priority messages are delivered. This is a typical effect of applying a very strict solution to avoid starvation. In the scenario described above, consider the following solution. The protocol may be modified to force the ordering of a message that is *older enough.* With such a solution, messages that have been waiting *too much* time are ordered as soon as possible. If there are some high priority messages ready to be ordered, they are forced to wait until low priority messages are ordered, (i. e., high priority messages have to wait because *priorities have been inverted*).

In [94, 16, 31], priority inversion is addressed when scheduling the access processes make to certain resources. In [101, 100], the *group priority inversion* is addressed in the context of actively replicated database applications that run in timed asynchronous systems. In that work, priority inversion affects complete user requests, rather than single messages. Priority inversion is also addressed in [6], in the context of real-time database transactions.

Part II

Prioritized Total Order

Chapter 3

Priority Management

In this chapter we present some techniques for adding priority management to existing total order protocols and show how can they be applied by means of pseudocode sketches. Moreover we briefly review some related work. The chapter is concluded with a revision of the *End-to-end Argument* in the context of the prioritization of total order protocols.

3.1 Adding Priority Management to Total Order Protocols

We have identified four basic techniques for adding priority management to total order protocols, mostly depending on the point in the life-cycle of the messages in which priorities are considered. These techniques may be called ***priority sequencing***, ***priority sending***, ***priority delivery*** and ***priority-based consensus***, respectively, and are explained in the following sections.

3.1.1 Priority Sequencing

Priority sequencing may be applied to sequencer-based total ordering protocols like ***fixed sequencer*** (Section 2.1.1) or ***moving sequencer*** protocols (Section 2.1.2). As the name suggests, the priorities of the messages are taken into account when the sequencer is about to sequence each message.

The idea is to keep a list of incoming items. These items may be the messages themselves (as in the fixed or moving UB and BB sequencer protocols) or requests to the sequencer to get a sequencer number (as in the fixed or moving UUB sequencer protocol). Prior to sending the item, the sender tags it with a priority. According to this priority, the item is inserted in its proper place in the list, so the list is ordered by priority.

To sequence the next item (a message or a request), the sequencer just gets the first available message in the incoming list, this is, the item with highest priority, and sequences them. The meaning of *sequencing an item* depends on the particular version (UB, BB or UUB) of the protocol, as explained in Sections 2.1.1 and 2.1.2.

This scheme is quite simple but low priority messages may undergo a starvation problem (see Section 2.4) that can be solved using a periodic timeout. When the timeout expires, the sequencer can block the reception of incoming items, assign sequence numbers to all the items currently in the list, broadcast all of them in decreasing order of priority and finally unblock the reception of incoming items. This way, low priority items have a chance of being finally sequenced.

The Cost of Priority Sequencing

The priority of an item (a message or a request) is just an integer so it does not impose a significant overhead in the size of the items and no significant overhead in the use of the network is appreciated.

On the other hand, there is a computational cost related to the reception of the items by the sequencer. Every time a new item is received by the sequencer, it must be inserted in its right place in the list of incoming items, which is ordered in decreasing order of priority. This insertion has a cost in time which is linear to the current size of the list, which in turn depends on several factors, like the sending rate of all the processes and the throughput of the sequencer (number of items sequenced per time unit).

3.1.2 Priority Sending

Priority sending differs from ***priority sequencing*** in that the priorities of the messages are taken into account in the moment the messages are sent. This kind of modification applies to ***privilege-based*** protocols (Section 2.1.2), some protocols of the ***deterministic merge*** subclass of the ***communication history*** protocol class (Section 2.1.4) like the one presented in Section 2.1.4 and the first class of ***destinations agreement*** protocols (see Section 2.1.5), presented in [39].

The idea behind this kind of modification is to send messages in a priority-based order. To this end, each node has a priority-ordered list of outgoing messages quite similar to the one used in Section 3.1.1.

Each outgoing message is placed in its right position in the list, according to its priority. Messages are taken from the head of the list and sent. They then may be treated according to the final protocol used to totally order the messages.

As messages are sent according to the order set by their priorities, if a sender has several outgoing messages pending to be sent, it may send first the most priority one. This also means that low priority messages are retained in the last places of the list, so a starvation problem may occur. A solution similar to the one proposed in 3.1.1 may be applied here if needed.

The Cost of Priority Sending

The cost of adding priorities by means of this modification is similar to that in Section 3.1.1.

3.1.3 Priority Delivery

The protocols of the ***causal history*** subclass of the ***communication history*** class of [39] can be modified to order messages according to the priorities of the messages.

In the original protocol, causal timestamps are used to causally relate messages. These timestamps are enough to totally order causally dependent messages. Concurrent messages, this is, those that are not causally dependent on each other, are totally ordered by means of a deterministic mechanism, that usually makes use of the identifier of the sender and the sequence number of the message (local to its sender).

The ***priority delivery*** modification proposed consists in taking into account the priority of concurrent messages prior to any other criteria. Note that causally dependent messages must still be ordered according to the causal relation imposed by their timestamps, in spite of their priorities, because the modified protocol must still provide the same causal and total order guarantees provided by the original protocol.

According to this, if some node broadcasts a low priority message and then broadcasts a high priority message, as the first one is causally precedent to the second one, the delivery of the latter must be delayed until the first is delivered, regardless of their priorities.

In the original protocol, causally dependent messages are totally ordered by means of their causal timestamps. Concurrent messages are ordered by means of a local deterministic mechanism, applied in delivery time. As the modification proposed is used to modify the way concurrent messages are totally ordered, it is also applied in delivery time.

To solve this issue, ***priority delivery*** can be combined with ***priority sending***. This way, messages are sent according to their priorities and as causality is enforced by the delivery mechanism, the local priority-based order is ensured. Moreover, according to the ***priority delivery*** modification, concurrent messages are also ordered according to their priorities.

The Cost of Priority Delivering

As in the previous modifications proposed, no significant overhead is imposed on message traffic. The computational cost is similar to that in other classes of protocols and basically depends on the message sending rates.

The proposed modification imposes no significant overhead in the size of the messages. The computational cost mainly depends on the message sending rate. Whenever a message is received by a processor, it is inserted in its right place in the list of incoming messages, depending on its timestamp, the priority and the sender identifier of the incoming message and the existing messages.

The cost of this insertion is linear to the number of messages currently in the list, but to be precise, part of this overhead was already present in the original version of the protocol (that corresponding to the comparison of the timestamp of the incoming message against the timestamps of other messages in the list).

Anyway, the size of the incoming list of a process directly depends on the sending rate of the processes in the system and the *productivity* of the local process (measured in messages locally ordered per time unit).

3.1.4 Priority-based Consensus

To end this classification of modification techniques, the ***priority-based consensus*** modification is presented, which is applicable to the second and third classes of ***destinations agreement*** protocols (see Section 2.1.5), presented in [39].

The modification, which is actually quite similar to that of Section 3.1.3, consists in taking into account the priorities of the messages, prior to other criteria, to reach the consensus about the order of a set of messages.

The Cost of Priority Consensus

This modification imposes no significant overhead on message traffic. The computational overhead is also low and, as in other cases, it directly depends on the message sending rates of all the nodes. Moreover, in order to perform the consensus, some additional memory space for message buffering purposes may be needed.

3.1.5 A Visual Classification

Table 3.1 shows, in a visual format, which modification type corresponds to each of the total order protocol classes (and subclasses) presented in [39]. It is also shown, for each protocol class, which is the agent that decides the total order.

	Modification				Who orders			
Protocol	PQ	PS	PD	PC	Q	S	L	O
Fixed UB	x				x			
Fixed BB	x				x			
Fixed UUB	x				x			
Moving UB	x				x			
Moving BB	x				x			
Moving UUB	x				x			
Privilege-based		x				x		
Comm. hist.-causal hist.			x				x	
Comm. hist.-det. merge		x					x	
Dest. agreement 1		x				x		
Dest. agreement 2				x			x	
Dest. agreement 3				x				x

Table 3.1: A visual classification of total order protocol classes

The *Modification* keys PQ, PS, PD and PC correspond to ***priority sequencing***, ***priority sending***, ***priority delivery*** and ***priority-based consensus***, respectively. The *Who orders* keys Q, S, L and O correspond to *sequencer*, *sender*, *local node* and *other*, respectively.

3.2 Algorithms

In this section four algorithms that implement the priority-based total order broadcast service are shown. Each algorithm corresponds to each of the four techniques presented in Section 3.1. These algorithms are just sketches and have not been formally proved. Instead, they just try to illustrate the four techniques of modifying existing total order broadcast protocols.

The algorithms shown are sketches that are not fault-tolerant, since fault-tolerance is not addressed in this work. Issues like node failures or network partitions are usually covered by other parts of a Group Communication System, like the underlying message transport layers and the group membership system. Our algorithms try to show the basic protocols and how they can be modified, in the same way of [40], discarding additional issues like fault-tolerance mechanisms. Nevertheless, any solution to address fault-tolerance in similar protocols may be easily applied to the sketches presented here. This approach allows us to compare how the basic protocols operate and perform, without the influence of additional tasks and mechanisms. This way we can focus on comparing the prioritization techniques and the resulting protocols.

In Algorithms 1, 2 and 3, a modification of the original fixed ***UB*** algorithm presented in [39] is shown (underlined text shows the main differences). The modification corresponds to the ***priority sequencing*** class of algorithms and it is actually very similar to the original fixed ***UB*** algorithm. The main difference is that incoming messages are not immediately sequenced and sent to all the destinations but queued according to their priority and later sent.

Algorithm 1 Modified fixed UB (sender)

```
Procedure TO-broadcast($m$, $\underline{prio}$):
    $\underline{prio(m) \leftarrow prio}$
    send $m$ to sequencer

```

Algorithm 2 Modified fixed UB (sequencer)

```
Initialization:
    $seqnum \leftarrow 1$
    $\underline{incoming \leftarrow \{\}}$

$\underline{\text{Parallel: when receive } (m)\text{:}}$
    $\underline{\text{insert } m \text{ in } incoming\text{, according to } prio(m)}$

$\underline{\text{Parallel: after initialization:}}$
    while $incoming$ is not empty do
        $\underline{m \leftarrow \text{first message in } incoming}$
        $\underline{incoming \leftarrow incoming \setminus \{m\}}$
        $sn(m) \leftarrow seqnum$
        send $(m, sn(m))$ to all
        $seqnum{+}{+}$
    end while

```

Algorithm 3 Modified fixed UB (destination)

```
Initialization:
    $nextdeliver \leftarrow 1$
    $pending \leftarrow \emptyset$

When receive ($m$, $seqnum$):
    $pending \leftarrow pending \cup \{(m, seqnum)\}$
    while $\exists(m, seqnum) \in pending : seqnum = nextdeliver$ do
        $deliver\ m$
        $nextdeliver{+}{+}$
    end while

```

A modification of the ***privilege-based*** algorithm of [39] is shown in Algorithms 4 and 5. This modification corresponds to the ***priority sending*** class of algorithms.

In Algorithm 6, the modification of the ***causal history*** algorithm presented in [39] is shown. This modification corresponds to the ***priority delivery*** class of algorithms. As the original algorithm, it assumes that a FIFO multicast algorithm is available.

Finally, Algorithms 7, 8 and 9 show a ***destinations agreement*** algorithm that fits into the third subclass identified in [39] and corresponds to the ***priority-based consensus*** class of algorithms. This sketch is presented just as an illus-

Algorithm 4 Modified privilege-based algorithm of process p (sender)

```
Initialization:
    $\underline{tosend \leftarrow \{\}}$
    **if** $p = s_1$ **then**
        $token.seqnum \leftarrow 1$
        send $token$ to $s_1$
    **end if**

Procedure TO-broadcast ($m$, $\underline{prio}$):
    $\underline{\text{insert } m \text{ in } tosend \text{ according to } prio}$

When receive $token$:
    **if** $tosend \neq \emptyset$ **then**
        $m \leftarrow$ first message in $tosend$
        send ($m$, $token.seqnum$) to destinations
        $token.seqnum++$
        $tosend \leftarrow tosend \setminus \{m\}$
    **end if**
    send $token$ to $s_{i+1\ mod\ n}$

```

Algorithm 5 Modified privilege-based algorithm (destination)

```
Initialization:
    $nextdeliver \leftarrow 1$
    $pending \leftarrow \emptyset$

When receive ($m$, $seqnum$):
    $pending \leftarrow pending \cup \{(m, seqnum)\}$
    **while** $\exists(m, seqnum) \in pending : seqnum = nextdeliver$ **do**
        deliver $m$
        $nextdeliver++$
    **end while**

```

Algorithm 6 Modified communication-history algorithm of process p (sender/destination)

```
Initialization:
    $received \leftarrow \emptyset$
    $delivered \leftarrow \emptyset$
    $LC \leftarrow \{0, \ldots, 0\}$

Procedure TO-broadcast($m$, $\underline{prio}$)
    $LC[p] \leftarrow LC[p] + 1$
    $ts(m) \leftarrow LC[p]$
    $\underline{prio(m) = prio}$
    send FIFO ($m$, $ts(m)$) to all

When receive ($m$, $ts(m)$):
    $LC[p] \leftarrow max(LC[p], ts(m)) + 1$
    **if** $\underline{p \neq sender(m)}$ **then**
        $\underline{LC[sender(m)] \leftarrow ts(m)}$
    **end if**
    $received \leftarrow received \cup \{m\}$
    $deliverable \leftarrow \emptyset$
    **for** each message $m$ in $received \setminus delivered$ **do**
        **if** $ts(m) \leq min_{q \in \Pi}\{LC[q]\}$ **then**
            $deliverable \leftarrow deliverable \cup m$
        **end if**
    **end for**
    deliver all messages in $deliverable$ in increasing order of ($ts(m)$, $\underline{prio(m)}$, $sender(m)$)
    $delivered \leftarrow delivered \cup deliverable$

```

trative example about how to apply this technique. According to this protocol, the messages received by the nodes are not directly delivered to the application but queued in a list. From time to time, a *starter* node decides to start a consensus round by sending to all the nodes a special START_CONSENSUS message. When a node receives such a message, sends a response that contains a set with the identifiers of the pending messages. When the *starter* receives a minimum number k of responses then it decides the common subset to all the received sets. The *starter* then decides a suggested total order of the selected messages, taking into account the priorities of the messages. This suggested order is sent to all the nodes (by means of an APPLY_ORDER message) so they can deliver the selected messages in the proper priority-based total order.

In the proposed algorithm, the *order* procedure is used to select the common subset of messages and to order them according to their priorities. The *vote* procedure is used to locally decide if the order proposed by the *starter* node is accepted. The voting mechanism is not addressed, but nevertheless, keeping this procedure apart from the *order* procedure makes possible to use any consensus [96] mechanism.

Algorithm 7 Modified destinations agreement, subclass 3 (starter)

```
1:  Initialization:
2:      $incomingIds \leftarrow \emptyset$
3:      $votes \leftarrow \{\}$
4:
5:  When start consensus:
6:      send $START\_CONSENSUS$ to all
7:
8:  When receive ($ids$):
9:      add $ids$ to $incomingIds$
10:     **if** $|incomingIds| \geq k$ **then**
11:         orderAndPropose()
12:     **end if**
13:
14: When receive (vote $v$):
15:     add $v$ to $votes$
16:     **if** there is a majority of positive votes **then**
17:         send $APPLY\_ORDER(orderedIds)$ to all
18:     **end if**
19:
20: Procedure orderAndPropose:
21:     $orderedIds \leftarrow order(incomingIds)$
22:     send $orderedIds$ to all
23:
```

Algorithm 8 Modified destinations agreement, subclass 3 (sender)

```
24: Procedure TO-bcast($m$, $prio$):
25:     $prio(m) \leftarrow prio$
26:     send $m$ to all
27:
```

Algorithm 9 Modified destinations agreement, subclass 3 (destination)

```
Initialization:
    $incoming \leftarrow \{\}$
    $ids \leftarrow \{\}$

When receive ($m$):
    add $m$ to $incoming$
    add $id(m)$ to $ids$

When receive ($START\_CONSENSUS$) from starter:
    send $ids$ to starter

When receive ($orderedIds$):
    $vote \leftarrow vote(orderedIds)$
    send $vote$ to starter

When receive ($APPLY\_ORDER(orderedIds)$):
    $ids \leftarrow ids \setminus orderedIds$
    while $orderedIds$ is not empty do
        $id \leftarrow$ first $id$ in $orderedIds$
        $orderedIds \leftarrow orderedIds \setminus \{id\}$
        $m \leftarrow$ message in $incoming$ with $id$
        deliver $m$
        $incoming \leftarrow incoming \setminus \{m\}$
    end while

```

3.3 Related Work

Unlike plain total order broadcast, priority-based total order broadcast has not been too much studied and few results have been presented. In this section, the most interesting results are commented.

In [84] (an extension of [83]), a starvation-free priority-based total order protocol is presented. The protocol sits on top of an existing total order broadcast service so the protocol in all the processes receives the messages in the same order. It then locally and deterministically orders messages according to their priorities.

Time is divided in *time parts*, and the protocol ensures that all the messages that belong to the same *part* are totally ordered according to their priorities. This time-based solution is also used to avoid starvation of low-priority messages.

The protocol keeps a queue of incoming messages that is ordered according to the priorities of the messages. Messages with the same priority are queued according to their arrival order. Incoming messages are queued in their corresponding place.

High priority messages are in the head of the queue and low priority are in the tail of the queue. To deliver messages according to their priority, messages are taken from the head of the queue and delivered to the application. To avoid starvation of low priority messages a periodic timer is used. Every time the timer expires, the reordering of incoming messages is temporarily paused. The protocol then forces the delivery of as many messages as possible, so low priority messages have a chance to be delivered to the application.

Such protocol is an extension of some existing total order protocol rather than a total order broadcast protocol itself and does not integrate the priority management in the total order protocol core. For this reason, it cannot be classified according to the taxonomy of [39].

In [87], another priority-based total order protocol is presented. This protocol guarantees that a message that has been received by all the processes will be delivered in the same order by all the processes, before any other message of a lower priority that has not been delivered by any process yet. To achieve this guarantee, a *Priority accounting* property is stated. According to this property, if a message of a given priority has not been delivered at any process, when a message of a higher priority is received, then the latter will be delivered prior to the former. Thus, a *priority-based total order multicast protocol* is defined as a broadcast protocol that preserves the *Priority accounting* property in addition to a regular *Total order* property.

The protocol keeps a list of incoming messages that is ordered according to the priority of the messages. This list has a common suffix in all the processes. As some processes are faster than others, they deliver messages to the application faster than others, so the head of the list is, in general, different.

When a message is sent, all the processes receive it (unless they fail). When a process receives a message, it blocks part of the list of the incoming messages (the part that contains messages of a lower priority). The process then sends some information related to the blocked part of the list of incoming messages to a special process that acts as a coordinator. It also sends information about the last messages delivered to the application. The coordinator uses all this information to decide in which point of the list processes must insert the incoming message.

This protocol undergoes starvation of low priority messages if too many high priority messages are sent. According to the authors, this protocol is especially suitable "for state machine-like applications where the time to consume a message is far greater than the time involved in the communication rounds" so no starvation issues are expected to appear, no solution is given nor even the problem itself is pointed at.

This problem may be solved as suggested in previous sections. From time to time, regular prioritization may be blocked, and lower priority messages are given a chance to be delivered. As previously said, this solution may cause priority inversion, as high priority messages may have to wait for low priority messages.

Regarding the classification of [39], as the delivery history is used to decide the order of the messages (as well as the priorities of the incoming and existing messages), this protocol may be classified in the ***deterministic merge*** subclass of the ***communication history*** protocol class.

In [16, 101] another common problem of this kind of protocols, known as *priority inversion*, is addressed.

3.4 Discussion About the End-to-end Argument

The end-to-end argument [90, 80] is a design principle that can be applied to many different kinds of systems. According to this principle, functionality used by an application and often packed as a library or any other external form (e. g. as a *service* somehow offered by the operating system) is better placed in the application level. Several reasons are given in favor of the end-to-end argument and against the opposite *low-level* argument.

Regarding prioritized total ordering, the end-to-end alternative means taking into account message priorities at the application level while the low-level alternative means considering the priorities at the group communication system level, as presented in Section 3.1.

In the end-to-end alternative, the application tags its messages with a priority level (as in the proposed modifications in Section 3.1). Each node may send its messages according to their priorities. The messages are broadcast and totally ordered by means of a regular group communication system, and delivered to the application (in every node) according to a total order.

As this order does not reflect any priority order, the application is in charge of reordering the incoming messages according to their priorities. This reordering must be done in such a way that the total order property of the sequence of delivered messages is kept.

The reordering can be done in a distributed manner, by using some additional message rounds, which increase the cost of the priority-based total ordering service.

The reordering can alternatively be done in a local manner, by means of a deterministic method. In a first alternative, time can be logically divided as in [84]. Some special control message, periodically sent by some *coordinator* node and totally ordered respect regular messages can be used to decide the duration of the *time parts*. Messages in each time period can be locally ordered (also guaranteeing the total order property).

A second alternative consists in forcing each node to wait for a predefined number of incoming messages and then deterministically reorder them according to their priorities. This way, the control messages used in the first alternative are avoided.

Such end-to-end approaches offer the advantage of being compatible with any total order broadcast protocol. In [90], another advantage of such application-level kind of solutions is argued. An application may have some information which is not accessible by a lower level and this information may allow the application to tune the operation of the service in order to get better performance numbers. Regarding the prioritization of total order broadcast protocols, there is no special information accessible only to the application[1] that could be used

[1]Keep in mind that we are not comparing the use of an application-level priority-based

to get a better performance so, in our case, this argument is actually not a real advantage.

On the other hand, these application-level solutions have some disadvantages that must also be considered. First of all, both solutions impose a delay in the delivery of the incoming messages (until the next control message arrives or the expected number of messages are received) which depends on the duration of each *time part* or the number of required messages.

Furthermore, in case a node fails and later recovers it must ask another node for the messages it has missed. If the priority management is performed at the application level, the application is in charge of keeping a list of recent incoming messages in order to forward them to the recovering node. In the first solution, each node must save, at least, all the messages from the last control message. In the second case, each node must save all the messages received since the last reordering took place. In both cases, this need complicates the design and implementation of the application. Instead, a middleware-based alternative, like any of the modifications proposed in Section 3.1, offers a simple and powerful solution that allows the application designer to focus on the relevant aspects of the application.

Besides that, these application-level solutions are not much reusable. If no special efforts are made to keep this priority-based reordering in a modular component, then the solutions are not reusable at all.

Moreover, the design and the implementation of the application is more complicated, as pointed above. Indeed, the application not only must reorder messages according to their priorities but also respect the total order property of the original sequence of incoming messages. Application designers need to worry about concerns that are not directly related to the application core.

So, unless application designers very carefully design these non-core aspects of the application, there is a risk that additional software bugs are put in. Instead, if this functionality is designed and implemented by specialized designers, this risk can be highly reduced.

For all these reasons, the conclusion about the end-to-end argument is that, in this particular case, is not worthy at all to move to an application-level layer the priority-based (re)ordering of incoming totally ordered messages.

reordering of an existing total order service against a regular total order broadcast service but against a modified total order broadcast service that takes into account message priorities.

Chapter 4

Effectiveness of the Prioritization Techniques

In this chapter we present some experimental work we have done in order to compare original and modified total order protocols. First of all, the environment and the application used to do the tests are described. Then, we describe the methodology followed to run the tests as well as the parameters used. Finally, we present some figures and discuss the results.

4.1 Environment

To test the original and the prioritized protocols, we have designed a simple application which will be described later. The application is run under different configurations that will also be described later.

This application uses the services of a total order protocol which in turn uses a reliable transport layer. This layer is our own implementation of the sixth transport protocol presented in [95]. It is based on the services provided by an unreliable transport we built on top of the bare UDP sockets provided by the Java platform.

The experiments have been conducted in a system of four nodes with an Intel Pentium D 925 processor at 3.0 GHz and 2 GB of RAM, running Debian GNU/Linux 4.0 and Sun JDK 1.5.0. The nodes are connected by means of a 24-port 100/1000 Mbps DLINK DGS-1224T switch that keeps the nodes isolated from any other node, so no other network traffic can influence the results.

4.2 Test Application

We have designed a test distributed application that simulates the usage of a replicated database in which some integrity constraints have been defined. This application issues transactions that modifiy a value that is shared among all the instances (or *nodes*) of the distributed application. The application uses a total order protocol to broadcast the *writesets* of the transactions. Once the writesets are received by the instances, they evaluate the integrity constraints. If all the constraints are fulfilled, the writeset is applied, thus modifying the shared data. Otherwise, the corresponding writeset is discarded.

The total order protocol ensures that every instance of the application receives the same sequence of messages. As every instance takes the same decision regarding the acceptance or rejection of each writeset, every instance applies the same sequence of changes to the shared data, thus keeping it consistent among all the instances. Moreover, the use of a prioritized total order protocol allows the application to *reorder* the sequence of writesets to be applied by means of reordering the messages used to carry those writesets. This can be used by the application to reorder the sequence of transactions to apply in order to fulfill some application-level requirement.

Next we describe the test application in more detail. The test application keeps track of the overall amount of money being managed by a stock trade company for all its stock investors. Each broker of the company runs its own instance (node) of the application and operates on the stock exchange on behalf of the stock owners and a potentially large number of investors.

The application is composed of two main components. A first component continuously analyzes the stock market and suggests the most interesting options (stock selling or purchasing) according to different factors. A second component keeps track of the global balance of the stock trade company. When a broker decides to perform some operation suggested by the first component, the second component verifies the operation and applies the required updates to the global balance. If the operation implies the purchase of shares, the second component must check that it can be performed, considering the price of the purchase and the current global balance of the company. In some cases, this component rejects an operation (for instance, when the price of the purchase exceeds the global balance of the company).

As there are a number of brokers working for the company buying and selling shares, the global balance is continuously updated accordingly to the operations. In order to guarantee that the current value of the company's balance is consistent among all the nodes of the application, a total order protocol is needed. The total order protocol is used by all the nodes of the application to multicast the updates that are being applied so all the brokers see the same sequence of operations and apply the same sequence of updates to the global balance. This way, in every moment the global balance is consistent among all the nodes of the application.

We use a `BalanceTest` application to simulate the second component. The application is composed of a number of concurrent threads, each one representing a node managed by a different broker. Each node creates and broadcasts a number of messages, each one representing an operation (stock selling or buying) that may update the current balance. Each update carries an integer value. A positive value represents a stocks selling operation and the number is an increment to be applied to the global balance. A negative value represents a stocks purchasing operation and the number is a decrement on the global balance. To simplify the analysis of the results, we adopted the following convention regarding the range of the updates of the balance the application will allow. The integer values belong to a range whose upper limit is 1000. The lower value of the range can be parametrized, as we will show in Section 4.4. The value assigned to each message is computed by a random generator. Additional details about the seeds used in the tests will be given in Section 4.3.

All the messages are multicast to all the nodes using a total order protocol so all the messages are delivered by all the nodes in the same order. Nodes apply messages in the order they are delivered by the total order protocol. Applying a message means updating the global balance kept by the node. As all the nodes receive the same sequence of updates, all of them keep consistent their corresponding copy of the balance.

As described below, each node multicasts a sequence of messages, each one representing an update of the global balance corresponding to a stocks operation and its priority. This way we can simulate the normal operation of a regular stock trade company that has a number of brokers performing stock operations on behalf of several hundreds of investors.

A stock market is a very dynamic scenario in which a decision (for instance, to buy a number of shares by a given price) applied out of time can cause disastrous results. Due to the complexity of the stock markets, some decisions are more urgent than others, and, in some cases, must be prioritized over others. For this reason, a stock trading application needs some mechanism that allows the brokers to set the priority of the operations they are issuing, in order to get the highest benefits from them.

In our test application, each message also carries a second integer value that represents the priority of the message. The priority of a given message expresses the urgency of the corresponding operation. In a real stock trade application, these priorities are usually computed taking into account a big number of factors, like the status of the market, the recent evolution of the shares, some long-term histories, the risk of the operation (for stocks purchasing) or the expected benefit (for stocks selling), among many others.

In our test application we considered a simplified approach that easies the design and implementation of the application and also the analysis of the results. The priority is computed deterministically from only the value of the operation (purchasing or selling). Given the value v corresponding to an operation, its priority p is computed as $p = 1000 - v$. According to this expression, an update of the global balance with a value of 1000 has a priority of 0 and an update with a value of -1000 has a priority of 2000. Taking into account that priority management in the modified total order protocols is implemented considering a *the lowest value, the highest priority* rule, then the first update has a higher priority than the second one. In other words, we are prioritizing positive updates (from a share sale) over negative updates (from a share purchase).

In `BalanceTest`, we implemented another rule to *discard* some negative updates. When an update is about to be applied, the new balance is computed. If the new balance is greater or equal to zero, then the update is applied. If the new balance is negative, then the update is *discarded*. In other words, we do not allow the global balance to be in the red. This rule is used to show the differences between conventional (non-prioritized) and prioritized total order protocols.

4.3 Methodology

The expected behavior of an execution of `BalanceTest` depends on the conventional or prioritized type of the total order protocol used. When using a non-prioritized protocol, the nodes apply approximately the same number of positive and negative updates. When using a prioritized protocol, positive updates (sales) are prioritized, as stated before. This means that the balance kept by the nodes will be increased faster than decreased and less negative updates will be discarded.

To compare a prioritized total order protocol against the non-prioritized version of the same protocol, we run the `BalanceTest` application and count the number of updates that have been discarded in both versions. A number of messages discarded by `BalanceTest` using a prioritized protocol lower than the number when it uses the corresponding non-prioritized protocol means that the prioritized protocol has been able to prioritize a number of messages. The higher the difference is, the best is the prioritization achieved by the prioritized protocol.

To test the proposed techniques, we tested different protocols. For each protocol, we varied the number of messages broadcast by each node. We also tried different values for the lower bound of the numeric value of the updates. For each combination of these parameters, we executed the `BalanceTest` and got the number of updates discarded. The concrete values of these parameters are discussed in Section 4.4.

As usual, a single execution of `BalanceTest` is not reliable enough to test a prioritized protocol against its non-prioritized version. To get reliable results, each execution is repeated a number of times. Each execution yields a number of discarded updates and we can compute the mean and median values of all the executions of a given test. We then compare the mean and median numbers of discarded updates by both versions of a given total order protocol.

Finally, we tried to avoid that the sequence of messages sent by each node had influence on the results. We forced each node to send the same sequence of messages (i.e. the same sequence of priorities) in each test run with the same combination of the parameters and each protocol. This way, in the same test (combination of the rest of the parameters) all the protocols receive the same sequence of priorities, which allows us to notice better the differences among the protocols. As said before, each test is run a number of times, but it is not necessary to send exactly the same sequence of messages (priorities) in each execution of the series. The only really needed is to guarantee that in the i-th execution of a given test (combination of the parameters), a given node sends the same sequence of messages with all the protocols.

4.4 Parameters

In this section we describe the values of the parameters used to test the protocols. First of all, we describe a set of *fixed* parameters, whose values are the same for all the tests. Then we explain a set of *variable* parameters.

Each test is run by a single Java Virtual Machine that runs a single `BalanceTest` instance. This instance spreads four threads, representing four nodes. Each thread creates a sequence of messages, as described above, and sends them using a fixed sending rate (currently, 50 messages per second).

Each message is tagged with a priority value that ranges between a fixed maximum value equal to 1000 and a variable minimum value, described below.

The variable parameters are the protocol type, the number of messages sent by each node and the minimum bound for updates.

Regarding the protocols, we have implemented three non-prioritized total order protocols and then modified them according to the techniques proposed in Section 3.2 to get the corresponding prioritized protocols. The ***UB*** protocol is an implementation of the UB sequencer-based total order algorithm proposed by [55][1]. The ***UB_PRIO*** protocol is the corresponding prioritized version of ***UB***. The ***TR*** protocol implements a token ring-based algorithm, which is, in essence, similar to the ones of [82] and [10][2]. The ***TR_PRIO*** protocol is the corresponding prioritized version of ***TR***. Finally, the ***CH*** protocol is an implementation of the causal history algorithm shown in [40] and the ***CH_PRIO*** is its corresponding prioritized version.

We have executed different tests in which each node receives 400, 2000 and 4000 messages, respectively.

Moreover, we have used two different values for the minimum lower bound of the range for the integer values that represent the updates. The values used

[1]UB stands for *Unicast-Broadcast*, as in [40].

[2]In the ***TR*** protocol, when a node receives the token, it broadcasts just a message, as in [38], instead of broadcasting multiple messages, as in [82] and [10].

have been -1000 and -1200. Thus, the range for the updates are $[-1000, 1000]$ and $[-1200, 1000]$, respectively.

The combination of these values yields a total of 36 different settings. For each setting, we have ran 500 executions of the `BalanceTest` application.

4.5 Results

For each execution of the `BalanceTest` application we got the number of discarded messages. For each series of executions, we have got a series of 500 numbers of the discarded messages in each of those 500 executions. Then we got the mean and median values of each series and represented the medians graphically.

To represent the medians, we have divided them in two different groups, depending on the value used as a lower bound of the range for the integer values that represent the updates. In the first group, the medians correspond to a lower bound equal to -1000, while in the second group, the lower bound is equal to -1200. In Figures 4.1 and 4.2 we show the medians for the first and second group, respectively. In the X axis we represent the number of messages received by each node (400, 2000 and 4000 messages). In the Y axis we represent the number of discarded messages. The value represented is the median for each series of 500 executions.

4.6 Discussion

The experiments show that the prioritization techniques yield good results.

When the lower bound of the balance updates is -1000, the prioritized versions of the ***UB*** and ***TR*** protocols offer a very important reduction on the number of discarded messages, respect their original counterparts. Nevertheless, the reduction is lower in case of the ***CH_PRIO*** protocol respect to the original ***CH*** protocol. When the lower bound is -1200, the reduction is lower but still important in case of the ***UB*** and ***TR*** protocols.

In Table 4.1, we summarize the reduction we got in each case (in percentage values), considering different lower bounds and number of messages received per node.

As shown by Figures 4.1 and 4.2 and Table 4.1, the reduction offered by the ***CH_PRIO*** protocol is negligible. As explained in [70, 72], the original ***CH*** protocol offers a message delivery service with total and causal order guarantees and the modified protocol must necessarily offer the same guarantees. This means that the modified protocol must ignore message priorities when reordering and delivering causally dependent messages and can only take into

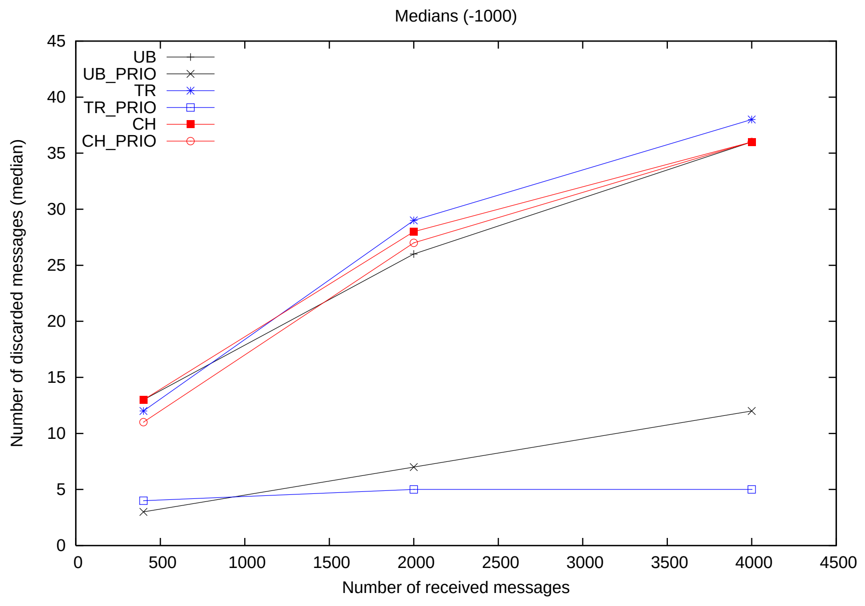


Figure 4.1: Median numbers (lower bound equal to -1000)

	-1000			-1200		
	400	2000	4000	400	2000	4000
UB_PRIO	76%	73%	66%	35%	24%	20%
TR_PRIO	66%	82%	86%	30%	28%	28%
CH_PRIO	15%	3%	0%	1%	2%	1%

Table 4.1: Percentage abort rate reduction

account message priorities to order and deliver concurrent (causally independent) messages. As the number of causally independent messages is small, the prioritization mechanism included in the ***CH_PRIO*** protocol can only offer a very small improvement respect to the original ***CH*** protocol.

Regardless the protocol, the lower bound of the balance updates also has a high influence on the results. When the lower bound is equal to -1000 the number of discarded messages is significantly lower, regardless the protocol used and the number of received messages per node, respect to the same setting run with a lower bound equal to -1200.

In the second case, the interval is $[-1200, 1000]$, which means that negative values are more likely than positive ones and therefore withdrawals are more likely than deposits. As there are more withdrawals than deposits, the balance gets smaller and smaller and withdrawals have a higher probability to be discarded.

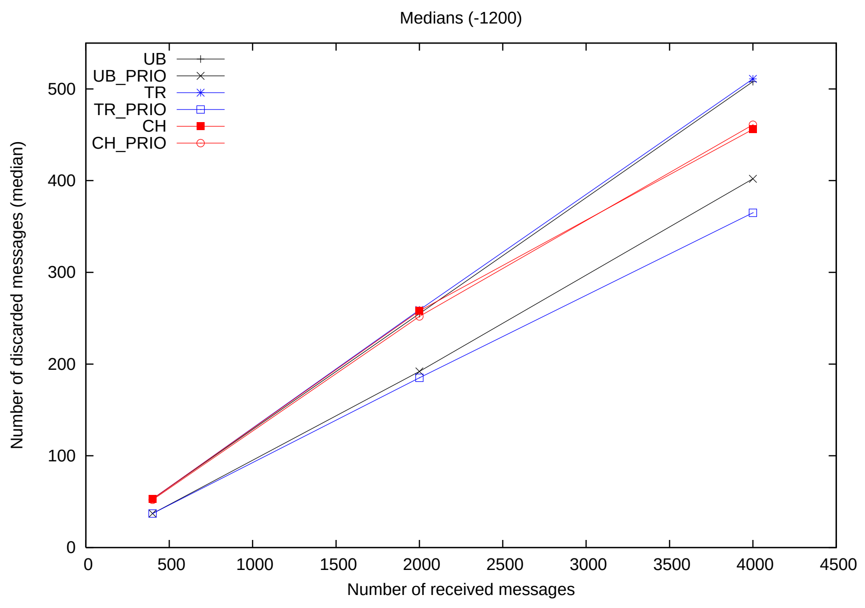


Figure 4.2: Median numbers (lower bound equal to -1200)

Nevertheless, in the first case, the interval of possible values for a balance update (deposit or withdrawal) is $[-1000, 1000]$. Positive and negative values have the same likelihood and therefore the number of discarded messages is lower than in the previous case.

There are other factors to consider when analyzing the effects of the prioritized protocols. The most determining factor is probably the application and the use it makes of the system. First of all, to benefit from a prioritized total order protocol, an application must send messages with different priorities. Moreover, prioritized protocols are only advantageous when there is a sustained flow of prioritized outgoing messages, sent at a minimum sending rate. If an application sends prioritized messages with a low sending rate, the prioritized messages are quickly handled, ordered and delivered, without colliding with other prioritized messages and no significant benefit is got from the prioritized protocols.

The results show that the application benefits from using prioritized versions of total order protocols. Current group communication systems do not include prioritization support in their total order protocols. The present results confirm that applications can benefit by making clever use of priority options offered by augmented versions of standard total order protocols. It is likely that existing group communication systems may be improved by adding, to their total order protocols, some prioritization support based on our techniques. The main contribution of this experimental study is to have shown that prioritized total order protocols are beneficial for applications that prioritize their transactions by taking into account their likelihood of violating given constraints.

Chapter 5

Cost of the Prioritization Techniques

In this chapter, we present the experimental work we have done to observe the performance of the total order protocols and evaluate the cost overhead of their prioritized versions. First of all, we describe the testbed, including the physical setting. Then, we describe the parameters and the methodology used to run the tests and finally we present and discuss the results.

5.1 Environment

To evaluate the prioritization techniques, the original and modified total order protocols described in Section 4.1 have been used. The system used to execute the experiments is also described in Section 4.1.

5.2 Methodology

To evaluate the performance of the prioritization techniques, in each node, the application broadcasts a series of messages to all the nodes in the system, by means of a total order protocol. The messages are broadcast at a uniform sending rate which is constant during the whole test. We have performed tests with different sending rates. Besides this, we have no other flow control mechanism neither in the application nor in the total order protocols.

Each message is tagged with a uniformly-distributed random priority which is an integer number.

The length of the messages is not fixed, but depends on the headers saved in them by the total order protocols. Nevertheless, in all the cases it is less than the

MTU of the network we are using (1500 bytes), so all the application messages fit into one wire-level packet.

Each message is totally ordered and delivered by all the nodes in the system. To evaluate the performance of a given protocol, we measure the *delivery time* of each message, i.e., the time observed by the application in a given node, from the moment in which it broadcasts the message to the moment in which it receives back the message, once totally ordered.

For each message we have a delivery time and for each node we have a series of delivery times, corresponding to all the messages sent by that node. If we merge all the delivery times from all the nodes, we can compute a global mean and median delivery time. Such a mean (median) time expresses the mean (median) time needed by messages to get totally ordered.

This test is run with different total order protocols and also with their corresponding prioritized versions. With these values we analyze the dispersion of the series of delivery times. A significant difference between the mean and the median values, especially when the median is lower than the mean, implies that there is a number of (low priority) messages that have a high delivery time, which means that the prioritization mechanism is working as expected and has been able to prioritize a number of messages. Nevertheless, the mean value of the test should not exceed some bound. An excessively high value for the mean delivery time implies that too many messages are being delayed and this delay is extending their delivery times. In this case we say that the protocol became *saturated*.

In order to get more trustworthy results, we discard the first 3200 messages[1] recorded in each node. These values correspond to delivery times of messages delivered during a period of time in which the total order protocol is being initialized so the system is not yet in a steady-state regime.

During the execution of these tests we also analyzed two additional indicators: a) the processing time employed by the prioritization mechanisms and b) the memory use. In Section 5.3 we provide additional details.

5.2.1 Parameters

The considered parameters are the class of total order protocol, the number of nodes and the sending rate at which the test application broadcasts messages.

Protocol type. To perform the evaluation, we have used the three non-prioritized total order protocols (***UB***, ***TR*** and ***CH***) and their corresponding prioritized versions (***UB_PRIO***, ***TR_PRIO*** and ***CH_PRIO***) described in Section 4.4.

[1]This number has been chosen empirically, after analyzing the behavior of the data structures managed by the total order protocol implementations.

Sending rate. In each test, a node broadcasts messages using a uniform sending rate. We have run tests with 4 and 8 nodes and sending rates of 10, 40, 60, 80 and 100 messages sent per second and node. Note that this generates maximum global sending rates of 400 msg/s and 800 msg/s, in systems with 4 and 8 nodes, respectively.

Number of messages delivered by each node. To ease the comparison, in each test, each node receives the same sequence of messages. This sequence has 32000 messages. A test ends when all the nodes deliver those messages.

To ensure a stable operation of the protocols during a test, each node sends more messages than those strictly necessary. For instance, in a test with 4 nodes, each node would only need to send 8000 messages. In practice, as the nodes deliver messages at a rate lower than the sending rate, there is a final period of time in a test in which the system is no longer *stable*, because the queues of the protocols are getting empty and this may affect the measuring of the delivery times. Moreover, the difference between the sending rate and the delivery rate is different in each test, and depends basically on all the parameters (the protocol used in the test, the number of nodes and the sending rate itself) and this poses additional difficulties to the protocol comparison.

To solve this issue, each node sends as many messages as needed, to ensure a continuous flow of messages during the whole test. This approach also solves the lack of liveness shown by the ***CH*** and ***CH_PRIO*** protocols, as described in [40].

5.3 Cost Evaluation

To evaluate the cost employed by the prioritization mechanisms, for each original protocol and its corresponding prioritized version we measure the time employed to run certain parts of both protocols. We call this time the *prioritization time.* The sections measured are semantically equivalent, so we can get comparable measures.

For instance, to evaluate the sequencer-based protocols, we measure the time lapse between the time when the sequencer starts to handle a message and the time when it broadcasts the message, once sequenced. The corresponding prioritized protocol has an equivalent section, in which prioritization takes place. Measuring the time needed to run both sections and comparing both times, we can get a very tight approximation of the time needed by the prioritization mechanism applied by the prioritized protocol.

These measures are only comparable between a given protocol and its corresponding prioritized version. For other protocol families, the parts of the protocols considered are different.

Table 5.1: Delivery times (ms) with 4 and 8 nodes

		UB	*UB_PRIO*	*TR*	*TR_PRIO*	*CH*	*CH_PRIO*
		4 nodes					
10 msg/s	mean	1.45	1.25	6.69	6.33	76.77	77.00
	1st q.	1.20	1.08	0.89	0.89	65.13	65.16
	med.	1.28	1.18	1.26	1.28	81.10	81.35
	3rd q.	1.36	1.26	9.30	7.52	93.38	93.44
40 msg/s	mean	1.50	1.46	1.29	1.27	17.77	17.86
	1st q.	1.11	1.09	0.72	0.72	13.13	13.10
	med.	1.24	1.31	1.02	1.02	17.12	17.01
	3rd q.	1.34	1.54	1.27	1.27	20.84	20.84
60 msg/s	mean	1.30	1.51	1.70	1.70	12.22	11.95
	1st q.	0.97	1.09	0.75	0.76	8.83	8.77
	med.	1.09	1.32	1.07	1.08	12.60	12.61
	3rd q.	1.24	1.53	1.32	1.35	12.88	12.91
80 msg/s	mean	3.43	2.20	2.36	2.75	9.13	9.29
	1st q.	1.17	1.27	0.87	0.77	4.97	4.96
	med.	1.27	1.42	1.20	1.09	8.66	8.62
	3rd q.	1.53	1.70	1.51	1.37	8.98	8.96
100 msg/s	mean	134.25	487.36	4.85	26.14	7.10	6.71
	1st q.	1.12	1.35	0.83	0.83	4.62	4.59
	med.	1.28	1.6	1.17	1.18	4.84	4.83
	3rd q.	1.79	2.75	1.51	1.52	5.14	5.20
		8 nodes					
10 msg/s	mean	1.89	11.35	2.05	2.08	90.33	90.96
	1st q.	1.34	1.53	1.37	1.39	85.21	85.40
	med.	1.53	1.73	1.86	1.87	97.62	94.21
	3rd q.	1.70	1.92	2.39	2.4	101.79	101.74
40 msg/s	mean	3.84	221.37	7.85	7.60	23.62	23.20
	1st q.	1.40	1.65	1.53	1.50	20.76	20.74
	med.	1.62	1.93	2.19	2.17	21.18	21.17
	3rd q.	2.04	2.86	2.91	2.86	21.89	21.68
60 msg/s	mean	190.82	670.48	75.53	151.49	17.09	17.22
	1st q.	1.42	1.72	1.68	1.69	12.96	12.98
	med.	1.86	2.51	2.54	2.53	13.28	13.27
	3rd q.	3.65	9.94	3.69	3.60	17.07	17.05
80 msg/s	mean	6718.52	13608.62	460.35	750.16	86.96	136.80
	1st q.	6373.32	13.32	2.24	2.21	9.02	9.18
	med.	6660.33	604.56	3.8	3.70	9.88	13.80
	3rd q.	6882.63	24776.30	340.26	34.26	65.51	237.24
100 msg/s	mean	20102.49	25264.85	5477.03	5148.22	100.05	125.82
	1st q.	14290.93	104.60	5119.25	5.14	5.78	5.70
	med.	18435.50	17349.36	5517.79	65.87	9.27	9.64
	3rd q.	23159.88	47670.92	5891.15	6908.98	58.16	145.75

For each test, we measure the prioritization time in each node[2]. Then we compute the mean prioritization time as the mean for all the nodes. These numbers are presented in great detail in [73, 74] and discussed in Section 5.4.

To evaluate the memory use, we analyzed how much of the total amount of memory available by the Java Virtual Machine is being used during each test by each node. In [73] we graphically represent this evolution in several settings (in systems of different sizes, with different protocols and sending rates, as explained in 5.2.1). Moreover, for each test, we count the number of times the Java garbage collector has been run in each node and with all of them, we compute the mean number of garbage collection runs. These numbers are explained in Section 5.4.

5.4 Results

In Table 5.1 we show the mean and median global delivery times (in ms), as well as the first and third quartiles in systems with 4 and 8 nodes, respectively,

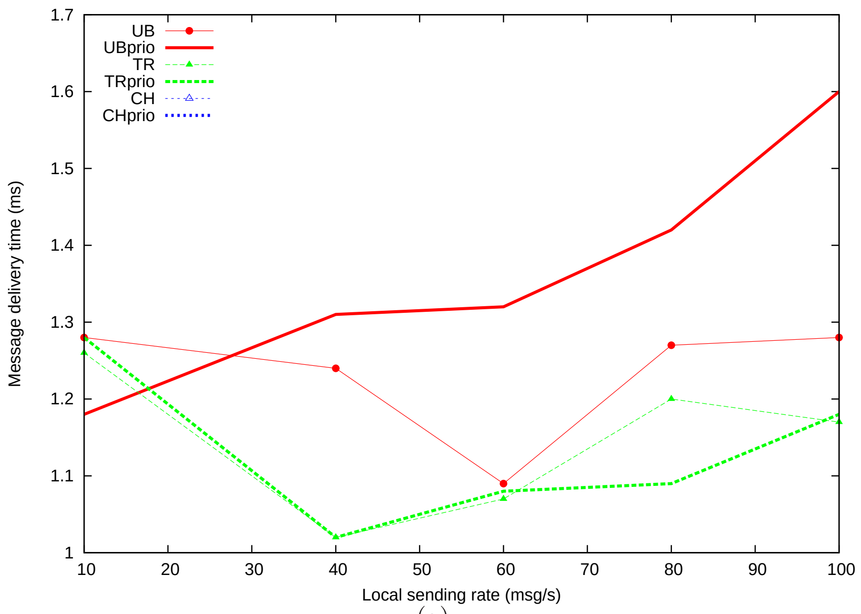
[2]We also discard the first 3200 messages, as explained in Section 5.2.

at different sending rates.

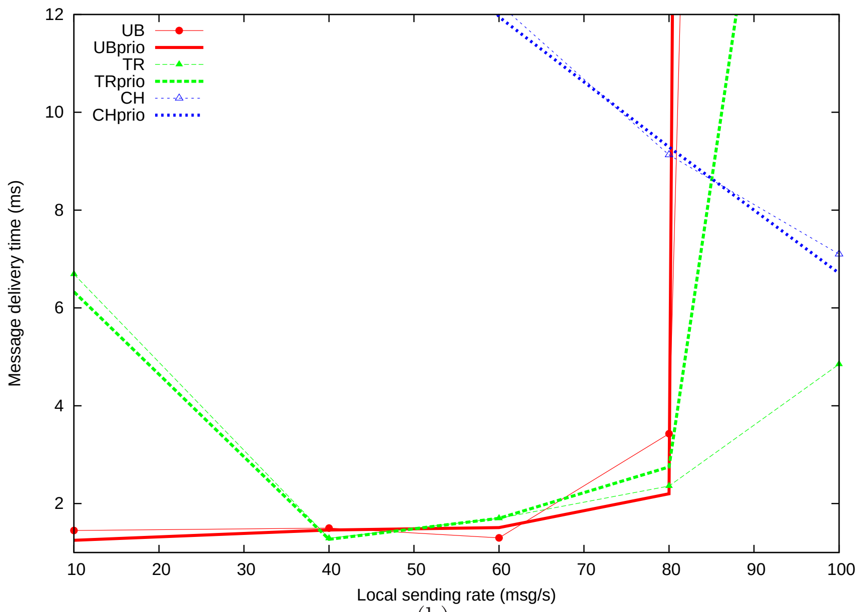
Delivery times in a 4-node system. In this configuration (see also Figure 5.1), the ***UB*** and ***UB_PRIO*** protocols perform well at sending rates up to 80 msg/s. At 100 msg/s ***UB*** still shows low median delivery times but their dispersion is high, because the protocol is getting saturated. This can be clearly seen in Figure 5.1.a (median times), where both protocols are able to deliver their messages in less than 1.6 ms, although ***UB_PRIO*** needs more time than ***UB***. Saturation is obvious when Figure 5.1.b is considered (mean times), since once the 80 msg/s threshold is surpassed, both protocols increase their delivery times with an exponential trend. Note also that there are no significant differences between both variants when their mean delivery times are considered.

The ***TR*** and ***TR_PRIO*** yield better performance numbers than sequencer-based protocols, even at 100 msg/s. At 10 msg/s the mean is slightly higher than expected although in these cases, the protocols are not saturated. When the sending rate is low, it may happen that the node which receives a token does not have any message to broadcast. In this case, it simply forwards the token to the next node in the ring. If a message is then broadcast by the application in the first node, then it will have to wait until the token arrives again to that node, thus increasing the delivery time of that particular message and also the mean delivery time. As this happens only to some messages, the delivery time of the rest of messages is low (due to the low sending rate and the low contention accessing the network). At higher sending rates this problem no longer arises. At 100 msg/s the dispersion in ***TR_PRIO*** is slightly higher as a side effect of the prioritization mechanism, as in ***UB_PRIO***. Despite this, it is able to scale quite better than sequencer-based protocols, since both its median and mean values are much lower than the latter ones.

Regarding the ***CH*** and ***CH_PRIO***, we can see that at low sending rates, the delivery time is very high but it decreases noticeably as the sending rate is increased. Indeed, no value has been shown for these protocols in Figure 5.1.a. On the other hand, they are the single family of protocols able to decrease its delivery time when the sending rate is increased. So, whilst sequencer-based and privilege-based protocols start to be overloaded in Figure 5.1.b, the communication-history ones start to be shown in such figure, presenting a descending trend. The design of the ***CH*** protocol forces an unordered message received by a node to wait until messages are received from the other nodes. Then, the order is locally (and deterministically) decided without any other message exchange. As the sending rate is increased, messages are forced to wait less time thus reducing the global mean and median delivery time. On the other hand, we can see that the dispersion is kept low in all the cases, since their mean and median always have close values. The reason of the delay experienced by the messages is mainly because ensuring the causal property imposes a delay on each message significantly greater than the delay imposed by the prioritization mechanism. As the delay imposed by the causal ordering is similar for all the



(a)



(b)

Figure 5.1: Median (a) and mean (b) delivery times in a 4-node system.

Table 5.2: Mean prioritization times (μs)

# nodes	msg/s	***UB***	***UB_PRIO***	***TR***	***TR_PRIO***
4	10	4.115	13.898	4.477	5.297
	40	4.034	9.834	4.255	3.504
	60	3.263	9.257	3.515	3.170
	80	3.520	9.834	3.717	2.175
	100	3.376	11.431	3.315	2.030
8	10	3.741	14.105	4.527	5.180
	40	3.547	12.120	5.129	4.833
	60	3.685	16.840	4.877	3.589
	80	3.644	15.255	4.794	5.024
	100	4.063	15.217	5.750	9.411

messages sent at a given sending rate, the dispersion of the delivery times is kept low.

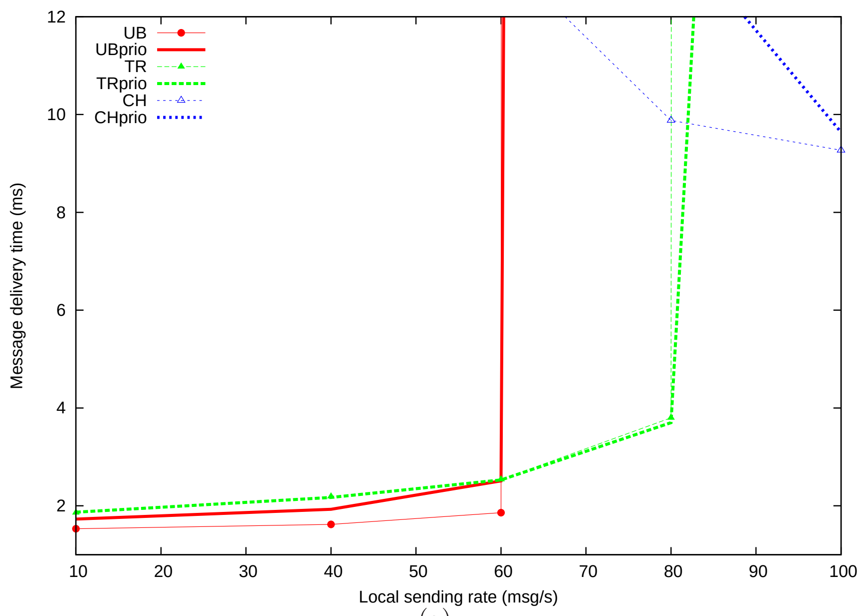
Summarizing, in Table 5.1 and Figure 5.1.b we can see that, in a system with 4 nodes, at sending rates up to 60 msg/s, the mean delivery time of any original (non prioritized) protocol is practically equal to the mean delivery time for the corresponding prioritized protocol, which means that the prioritization mechanisms are not imposing a noticeable overhead. Something similar happens to the median delivery times. Above 60 msg/s the numbers diverge because the load starts to be too high and then the response depends on each particular protocol, as already explained above.

Delivery times in an 8-node system. In such configuration (Figure 5.2.a), we can see that ***UB_PRIO***, and ***TR_PRIO*** offer good median delivery times at sending rates up to 60 msg/s. Moreover, these numbers are comparable to the ones for their corresponding original (non prioritized versions). At sending rates above 60 msg/s, these protocols get saturated, in varying degrees, and the delivery times start to get impractical.

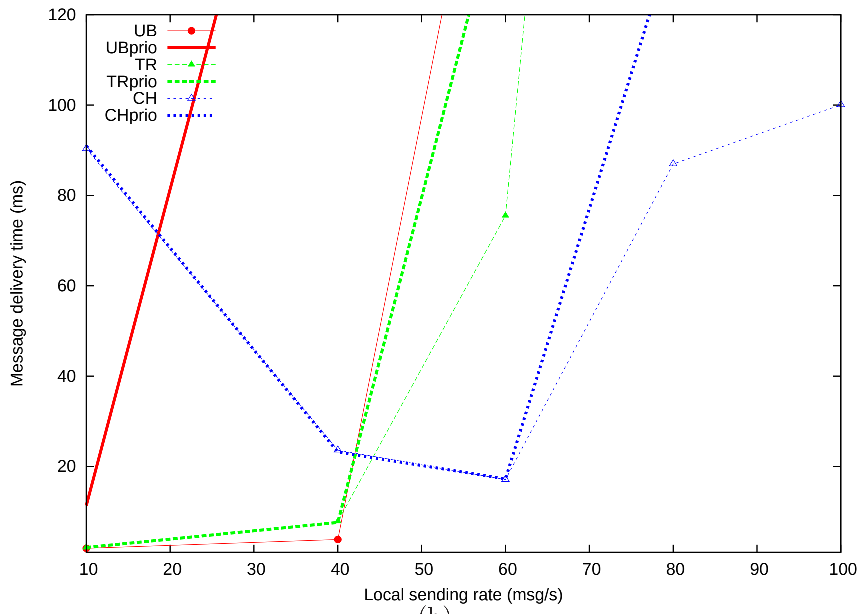
Regarding ***CH*** and ***CH_PRIO*** protocols, they show a similar trend to that found in a 4-node system. They are able to show a decreasing trend in their median delivery times and provide acceptable values at 80 and 100 messages sent per second. No other protocol is able to guarantee such delivery times at 100 msg/s.

Considering mean delivery times (Figure 5.2.b), the ***UB_PRIO*** protocol provides the highest increasing trend, starting with values at low sending rates that are only surpassed by communication-history protocols. However, since its median values are good up to 60 msg/s, this means that this family provides the best prioritization results in the range 10 to 60 msg/s, and that it gets completely saturated once such range is exceeded. Both privilege-based and communication-history protocol families stand much better for high sending rates, being ***CH*** and ***CH_PRIO*** protocols the clear winners.

Prioritization time overhead. Regarding the *prioritization times* presented in Table 5.2, we can analyze the differences among the values of the conventional



(a)



(b)

Figure 5.2: Median (a) and mean (b) delivery times in an 8-node system.

Table 5.3: Mean numbers of garbage collection runs.

# nodes	msg/s	***UB***	***UB_PRIO***	***TR***	***TR_PRIO***	***CH***	***CH_PRIO***
4	10	43.50	54.25	975.75	985.00	51.00	51.00
	40	27.00	30.75	62.00	62.50	41.25	42.00
	60	18.50	22.25	39.25	39.50	20.50	19.75
	80	17.25	26.25	31.50	30.75	17.00	15.75
	100	17.50	22.75	23.75	23.50	17.00	15.75
8	10	35.50	41.62	228.00	230.25	52.75	53.00
	40	18.62	25.12	25.00	24.75	18.62	18.87
	60	19.87	24.87	21.00	21.75	17.37	18.00
	80	22.12	25.50	19.87	20.12	18.00	18.37
	100	23.00	27.00	19.00	19.87	18.00	18.12

(non-prioritized) protocols and the prioritized ones. The bigger differences can be found when comparing the ***UB*** and the ***UB_PRIO*** protocols at any sending rate and with 4 or 8 nodes in the system. At a first glance it seems that the prioritization mechanism in ***UB_PRIO*** is introducing a significant load to the original protocol. Nevertheless, we can see that in all cases, the overhead is around a few microseconds, which compared to the full delivery time (in the order of milliseconds) is negligible.

In the case of the ***TR*** and ***TR_PRIO*** protocols, the differences are smaller, and again, compared to the full delivery times, are negligible. Moreover, in some cases the prioritized protocol is able to deal with these message management steps faster than the non-prioritized protocol. So, no overhead is introduced in this protocol family.

Finally, the ***CH*** and ***CH_PRIO*** protocols cannot be studied in this regard. The prioritization overhead is negligible in this protocol, since the message delivery time (once the message has been received in their target nodes) is highly dominated by the delays introduced in order to ensure causal delivery. Thus, the data to be shown in this table, according to the criteria used for other protocols, would have been approximately a 93% of the time shown in Table 5.1, but it would not provide the information shown in the other protocols (prioritization overhead), but mainly the causal-related delays.

Prioritization memory overhead. Memory usage is summarized in Table 5.3. To this end, such table presents the amount of garbage collections executed by the Java Virtual Machine in order to recover *free* memory that was previously assigned to dynamic data being used in the protocol. At a glance, in all cases a surprising trend is shown: the amount of garbage collections decreases when the sending rate is increased. This trend can be easily explained. Since the amount of messages being broadcast in each test is constant, the greater the sending rate is, the shortest the test length will be. So, there is nothing annoying in this behavior.

We can observe that in general, there are no big differences between the figures for the ***TR*** and ***CH*** protocols and the ones for their corresponding prioritized versions. This means that no memory-related overhead is being introduced by the prioritization mechanism in such protocols.

Significant differences exist however, among the numbers of garbage collection runs for the ***UB*** and those for ***UB_PRIO***. The reason of these differences is basically the memory overhead suffered by the sequencer node which typically uses more memory than the rest of the system nodes[3]. Note also that ***UB_PRIO*** has been the protocol providing the best prioritization results; i.e., for a particular workload, it was able to keep the median delivery time and both quartiles like the non-prioritized protocol, whilst its mean delivery time was quite longer. This means that there were multiple low-priority messages that needed a lot of time to be delivered. Such messages were kept in the sequencer queue, increasing the memory demands of such sequencer process, as Table 5.3 confirms.

On the other hand, it is also remarkable the high number of garbage collections required by both privilege-based protocols at the lowest sending rate (around 980 collections with 4 nodes, and around 230 with 8 nodes). This partially explains the long mean delivery time of such protocols in a 4-node system with a sending rate of 10 msg/s.

In [73], we depict the evolution of the amount of free memory available for the Java Virtual Machine during each test under different settings. The figures presented in Table 5.3 can be contrasted against those graphical representations.

Scalability. The best protocols in this issue seem to be the communication history ones, since they demand a high sending rate in order to provide acceptable delivery times. Thus, in a 4-node system their median delivery time starts with 81 ms at 10 msg/s and finishes with 5 ms at 100 msg/s. The same trend is shown in an 8-node system, but in the latter no improvement is detected form 80 to 100 msg/s. So, this protocol seems to start its overloading with a global sending rate of near 640 msg/s. Unfortunately, its prioritization quality is the worst one, since the mean delivery time in the prioritized variant is not longer than that of the non-prioritized one. This has been already explained: these protocols also guarantee causal delivery, and they can only prioritize concurrent (i.e., non-causally-related) messages.

The second best protocols, regarding scalability, are the privilege-based ones. They are able to serve individual sending rates of 100 msg/s in a 4-node system without any noticeable overload in both median and mean delivery times. This means that they have been able to comfortably deal with a global load of 400 msg/s in such system. In an 8-node system, they show the first signs of overloading at 60 msg/s (i.e., with a global load of 480 msg/s), with a mean delivery time of 151 ms in the prioritized version and 75 ms in the non-prioritized one, whilst the median delivery time was still below 3 ms. Despite this, the prioritized variant is able to maintain a median delivery time of 66 ms with a global sending rate of 800 msg/s.

[3]As stated in Section 5.2, these mean numbers are got from the numbers for all the nodes in the system, including its sequencer in case of the ***UB*** and ***UB_PRIO*** protocols.

The worst protocol family seems to be the sequencer-based one. Despite providing very good median delivery times in a 4-node system with all studied sending rates, it finds some problems to deal with the highest sending rate of such configuration (100 msg/s). In the latter case, its mean delivery times (134 ms in the non-prioritized version and 487 ms in the prioritized one) reveal that there were many messages with unacceptable high delivery times. The same starts to happen with similar global sending rates in an 8-node system (at 480 msg/s, mean delivery times exceed 190 ms in the best case), but delivery times get unaffordable values at 640 msg/s, quite longer than those of all other protocol families at 800 msg/s.

5.5 Conclusions

We have presented an experimental study in which we show that the prioritization techniques do not impose an important overhead (in terms of message delivery latency, processing time and memory use) on the original total order protocols, thus proving that, besides being easy to understand and implement, and being useful for replicated database management (as shown in [71, 76]), the techniques are affordable in terms of performance. The main conclusion is that prioritized total order broadcast protocols are a valuable building block that can be used to improve the design and implementation of distributed applications and their performance, as well.

As a second contribution this experimental study can be seen also as a performance comparison among conventional non-prioritized total order protocols. The results of this comparison show that sequencer-based and privilege-based protocols offer a comparable performance when the number of nodes is small (4 or 8) and the individual sending rate is not too high (around 60 msg/s). As the number of nodes or the sending rate is increased the sequencer-based protocols start to get saturated and the communication history ones improve their performance. At higher sending rates, communication history protocols are the unique ones that can stand such load.

Chapter 6

Dynamic Prioritized Total Order Protocol Replacement

The results presented in Chapter 5 show that there is no single total order protocol that provides the best performance under any working conditions. They also show that the prioritization techniques provide different results depending on several factors, like the system load.

In practice, this means that the election of a prioritized total order protocol may have a significant impact on the performance of the application. Specifically, the election of an *inappropriate* protocol may lead the application to get a worse performance. For this reason, the election of the protocol to use must be done carefully.

Nevertheless, there are some problems that must be considered. First of all, it is not easy to *guess* the working conditions an application will have, unless it is a very specific application that has already been carefully evaluated. On the other hand, even when the working conditions of the application are known beforehand, the election of the most suitable protocol requires from the designers of the application some knowledge about the available protocols. Moreover, it may happen that the working conditions of an application change during its execution, so the protocol first chosen as the most suitable becomes *unsuitable* due to these changes.

In practice, there should be some mechanism that allows the applications to use, in every moment, the most suitable prioritized total order protocol, according to different factors (application-dependent factors like the system load or message sending *patterns*, system-dependent factors like the underlying network and its topology, etc.). Moreover, such a mechanism should be *transparent* from the point of view of the protocols and the applications.

Such a mechanism offers several advantages. First of all, application designers do not need to *guess* the working conditions of the applications. They do neither need to know too many details about the protocols available nor about the best settings for each one of them. Moreover, such mechanism would allow an application to *adapt to* changing working conditions and, in general, to get a better performance.

6.1 A Dynamic Protocol Replacement Architecture

In [69], we presented an architecture for dynamically replacing Group Communication Protocols (GCPs). Such an architecture was designed to allow the dynamic replacement of FIFO or total order broadcast protocols.

In the following sections we review the work presented in [69, 77, 75] and discuss how such an architecture could be adapted to fit the needs presented above.

In Figure 6.1 we show a reviewed version of the high level graphical description of the architecture proposed in [69].

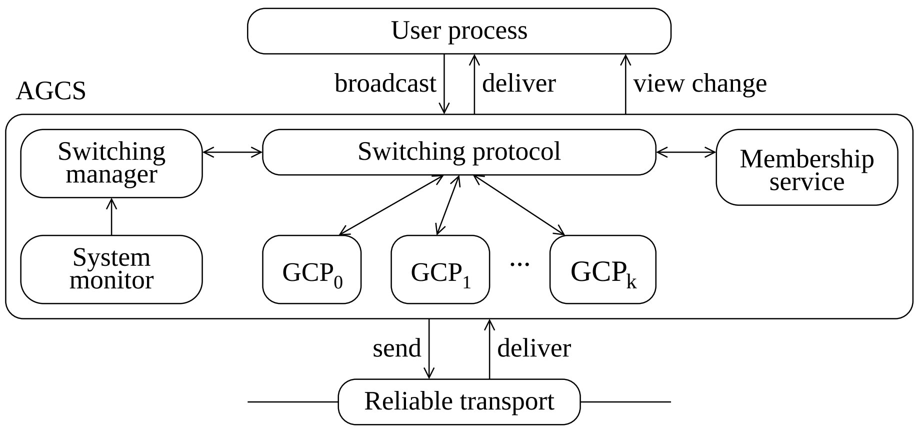


Figure 6.1: Architecture of a node

This architecture is composed of a main component, called *Adaptive Group Communication System* (***AGCS***). As shown in the figure, the user process sits on top of this architecture and this, in turn, relies on a regular reliable message transport layer. The ***AGCS*** wraps several standard group communication components (a number of GCPs, for instance, total order protocols and a membership service) and also specific components.

The ***Switching protocol*** implements the mechanism of replacing the GCPs in run-time. It captures the regular communication that occurs among the user process and the GCPs and performs the GCP replacement. The ***Switching manager*** is a component that decides when GCP changes should take place and which GCP should be installed. The ***Switching manager*** relies on a ***System monitor*** that keeps track of several system and application measurable variables and parameters. The ***Switching manager*** collects the measures provided by the ***System monitor*** and uses them to decide about GCP changes.

The original architecture presented in [69] and the one we present in Figure 6.1 include a ***Membership service*** component and a reliable transport layer. The ***Membership service*** provides notifications about changes on the set of nodes considered *alive* (due to joins of new nodes, node failures or node disconnections). Finally, the ***Reliable transport*** layer offers a regular *reliable* and *FIFO* message transport layer which ensures that a message sent to a destination is received by that destination unless it fails.

6.2 The Switching Protocol

In this section we present the ***Switching protocol***. We first provide an overview of the protocol and then we present some notation details and a pseudocode algorithm of the protocol. We finally discuss some details not covered in the first overview.

6.2.1 Overview

During normal operation, when no GCP replacement is being carried on, the ***Switching protocol*** takes charge of the messages sent by the user process, which are redirected to the current GCP. Incoming messages are received by the current GCP and directly handled to the protocol, which in turns delivers them to the user process. The core of the ***Switching protocol*** does not take part in this process.

A GCP replacement starts when the ***Switching manager*** instructs the ***Switching protocol*** in a particular node to start a GCP change. The ***Switching protocol*** in this *initiator* node *to–bcasts* a `PREPARE` message to inform all the nodes about the new change. At every node, the ***Switching protocol*** stops relaying messages with the current GCP, instances and initializes a new GCP and starts relaying messages with it. Moreover, each node *to–bcasts* a `PREPARE_ACK` message to tell all nodes about the number of messages it has sent with the previous GCP and waits for a `PREPARE_ACK` from all the nodes.

In the meantime, the ***Switching protocol*** goes on receiving messages delivered to it by the previous GCP and forwarding them to the user application. The ***Switching protocol*** may also receive messages delivered by the new protocol, as it has already been started in all nodes. These messages are not delivered to the user process yet, but queued in a local queue, until all messages broadcast with the previous GCP are delivered to the user process.

When the ***Switching protocol*** receives all the `PREPARE_ACK` messages it knows how many messages were sent with the previous GCP by each node. When all of them are finally received, the ***Switching protocol*** can finally discard the previous GCP. Then, it delivers to the user application all the messages broadcast with the new GCP, which were locally queued. When all of them are delivered, the ***Switching protocol*** can go on using the new protocol as the only available one.

The protocol receives view changes from an independent membership service, for instance, when a node failure happens. If such a notification is received during a protocol change, the protocol basically *stops waiting for* messages from the failed node, so the protocol change can proceed when a node failure happens. An additional discussion is provided in Section 6.3.3.

Moreover, the protocol is able to manage consecutive protocol change requests. The protocol ensures that if a protocol change request is received by a node while a previous request is being handled, the current protocol change is completed and the next one is then handled. Additional details are given in Section 6.3.2.

6.2.2 Pseudocode

The pseudocode algorithm of the protocol is shown in Algorithms 10 and 11.

The protocol uses several *global variables*. k is a counter of the GCP changes. It is initialized to 0 and incremented when a new GCP change is started. *changing_gcp* is a flag to know if there is a GCP change in progress or it has already finished. *live_nodes* is the set of live nodes as notified by the membership service.

The algorithm also uses a struct of type P for each GCP it manages. Thus, P_0 would be the struct for the first GCP used, P_1 would be the one for the second, etc. Such a struct contains several fields to store some state related to a GCP. Given a struct P_k, the expression $P_k.GCP$ is used to reference that GCP. The $P_k.k$ field is the number of the replacement by which the $P_k.GCP$ is installed. In general, $P_k.k = k$. $P_k.sent$ is the number of user messages that have been broadcast by $P_k.GCP$. $P_k.other\_sent$ is an array that stores the number of messages sent by all the processes in iteration $P_k.k$ by means of $P_k.GCP$. Each entry of the array is initialized to 0 and updated when a new protocol replacement is started, using the information received from each process. The number of messages sent by process q is $P_k.other\_sent[q]$ and it is initialized to 0. $P_k.delivered$ is an array that stores the number of messages sent by all the processes delivered by the local process by means of $P_k.GCP$. $P_k.delivered[q]$ is the entry corresponding to the messages sent by process q. Each entry of the array is initialized to 0 and updated by the local process each time it receives a message from $P_k.GCP$. $P_k.deliverable$ is a list of messages delivered to the protocol by $P_k.GCP$. If $P_k.GCP$ is not the current protocol but a later one, the messages delivered by it cannot be directly forwarded to the user process. Instead, they are stored in $P_k.deliverable$, until all the messages sent with all the previous GCPs are delivered.

We also assume that the managed GCPs provide a *to–bcasts* primitive to broadcast a message to all the nodes in the system. Given a message m, $m.sender$ denotes its sender.

The algorithm is composed by a set of *handlers* and *functions* which are executed as a response to external messages (sent by other nodes) and events (e.g. view change events produced by the ***Membership service***) or called from other event handlers and functions. These handlers and functions are *atomic*, i.e. we assume that two handlers or functions can not be executed concurrently.

The `INIT` function is executed only once, when the whole system is started. The `TO-BCAST` handler is invoked by the user application in order to broadcast a message (in total order). The `HANDLER_USER_MSG` handler is invoked by the GCPs to deliver incoming totally ordered messages to the ***Switching protocol***. The `START` function is executed when the ***Switching manager*** decides to start a new protocol change. The `HANDLE_PREPARE` and `HANDLE_PREPARE_ACK` are invoked by the GCPs to deliver `PREPARE` or `PREPARE_ACK` messages, respectively, to the ***Switching protocol***. The `FINISH_PENDING` function is invoked to try to finish as much pending protocol changes as possible. The `END` function is executed to finish a protocol change. The `HANDLE_VIEW_CHANGE` handler is invoked by some external membership service to deliver notifications on the membership view. The `DELIVERY_FINISHED` function is invoked to decide if all the pending messages needed to perform a protocol change have already been received.

6.3 Discussion

In this section we discuss some issues that were not covered in Section 6.2 to simplify the presentation of the protocol. These issues cover the normal operation of the protocol and also its behavior in presence of failures.

6.3.1 Normal Operation

The protocol we are presenting offers a number of advantages over the protocols reviewed in Section 6.6 and the protocol we proposed in [69].

First of all, our solution does not block the sending of user messages. When a node is instructed to start a protocol switch, the sending of messages with the current GCP is disabled but message sending is immediately enabled with the new GCP.

Algorithm 10 The ***Switching protocol*** pseudocode (part I)

```
 1: INIT($G$):
 2:     $current\_k \leftarrow 0$
 3:     $next\_k \leftarrow 0$
 4:     $changing\_gcp \leftarrow false$
 5:     instance, prepare and initialize $G$
 6:     call CREATE_P($P_{next\_k}$, $G$)
 7:
 8: CREATE_P($p$, $g$):
 9:     $p.GCP \leftarrow g$
10:     $p.k \leftarrow next\_k$
11:     $p.sent \leftarrow 0$
12:     $p.other\_sent[q] \leftarrow 0$, for each process $q$ in $live\_nodes$
13:     $p.ack\_received[q] \leftarrow false$, for each process $q$ in $live\_nodes$
14:     $p.delivered[q] \leftarrow 0$, for each process $q$ in $live\_nodes$
15:     $p.deliverable \leftarrow \{\}$
16:
17: TO-BCAST($m$):
18:     if $changing\_gcp == true$ then
19:         to-bcast $m$ with $P_{next\_k}.GCP$
20:         $P_{next\_k}.sent++$
21:     else
22:         to-bcast $m$ with $P_{current\_k}.GCP$
23:         $P_{current\_k}.sent++$
24:     end if
25:
26: HANDLE_USER_MSG($m$):
27:     if $m.k == current\_k$ then
28:         deliver $m$ to the local process
29:         $P_{current\_k}.delivered[m.sender]++$
30:         if $changing\_gcp == true$ then
31:             call FINISH_PENDING()
32:         end if
33:     else
34:         queue $m$ in $P_{m.k}.deliverable$
35:     end if
36:
37: START($G'$):
38:     to-bcast PREPARE($G'$) with $P_{current\_k}.GCP$
39:
40: HANDLE_PREPARE($G'$):
41:     $next\_k++$
42:     $changing\_gcp \leftarrow true$
43:     instance, prepare and initialize $G'$
44:     call CREATE_P($P_{next\_k}$, $G'$)
45:     bcast PREPARE_ACK($current\_k$, $P_{current\_k}.sent$) with $P_{current\_k}.GCP$
46:
47: HANDLE_PREPARE_ACK($k$, $sent$) from process $q$:
48:     $P_k.other\_sent[q] \leftarrow sent$
49:     $P_k.ack\_received[q] \leftarrow true$
50:     call FINISH_PENDING()
51:
52: FINISH_PENDING():
53:     $changing\_gcp\_aux \leftarrow false$
54:     for $j = current\_k$ to $next\_k$ do
55:         if $DELIVERY\_FINISHED(j)$ then
56:             call END($j$)
57:             $current\_k++$
58:         else
59:             $changing\_gcp\_aux \leftarrow true$
60:             break
61:         end if
62:     end for
63:     $changing\_gcp \leftarrow changing\_gcp\_aux$
64:
65: END($j$):
66:     for all $m$ in $P_{j+1}.deliverable$ do
67:         if $m$ is a user message then
68:             call HANDLE_USER_MSG($m$)
69:         else if $m$ is a PREPARE message then
70:             call HANDLE_PREPARE($m$)
71:         else
72:             call HANDLE_PREPARE_ACK($m$)
73:         end if
74:         remove $m$ from $P_{j+1}.deliverable$
75:     end for
76:     destroy $P_j.GCP$
77:
```

Algorithm 11 The ***Switching protocol*** pseudocode (part II)

```
78:  HANDLE_VIEW_CHANGE($failed\_nodes$):
79:      remove $failed\_nodes$ from $live\_nodes$
80:      call FINISH_PENDING
81:
82:  DELIVERY_FINISHED($j$):
83:      $totalOtherSent \leftarrow 0$
84:      $totalDelivered \leftarrow 0$
85:      for all $q$ in $live\_nodes$ do
86:          if $P_j.ack\_received[q] == false$ then
87:              return false
88:          end if
89:          $totalOtherSent += P_j.other\_sent[q]$
90:          $totalDelivered += P_j.delivered[q]$
91:      end for
92:      if $totalOtherSent == totalDelivered$ then
93:          return $true$
94:      else
95:          return $false$
96:      end if
97:
```

Algorithm 12 The ***Switching protocol*** pseudocode (part III)

```
98:  INIT($G$, $sending$):
99:      ...
100:     $provide\_sending\_view \leftarrow sending$
101:     $changing\_view \leftarrow false$
102:
103: TO-BCAST(m):
104:     if $changing\_view == true$ and $provide\_sending\_view == true$ then
105:         block call
106:     end if
107:     if $changing\_gcp == true$ then
108:         to-bcast $m$ with $P_{next\_k}.GCP$
109:         $P_{next\_k}.sent++$
110:     else
111:         to-bcast $m$ with $P_{current\_k}.GCP$
112:         $P_{current\_k}.sent++$
113:     end if
114:
115: HANDLE_VIEW_CHANGE($new\_nodes$, $failed\_nodes$):
116:     $changing\_view \leftarrow true$
117:     remove $failed\_nodes$ from $live\_nodes$
118:     to-bcast $NEW\_VIEW(new\_nodes, failed\_nodes)$ with $P_{next\_k}.GCP$
119:     call FINISH_PENDING()
120:
121: HANDLE_NEW_VIEW($new\_nodes$, $failed\_nodes$):
122:     add $new\_nodes$ to $live\_nodes$
123:     for all $q$ in $new\_nodes$ do
124:         for $j = current\_k$ to $next\_k$ do
125:             $P_j.other\_sent[q] \leftarrow 0$
126:             $P_j.ack\_received[q] \leftarrow false$
127:             $P_j.delivered[q] \leftarrow 0$
128:         end for
129:     end for
130:     deliver ($new\_nodes$, $failed\_nodes$) to the local process
131:     if $provide\_sending\_view == true$ then
132:         unblock call to TO-BCAST (if any)
133:     end if
134:     $changing\_view \leftarrow false$
135:
```

Moreover, it allows both protocols to coexist and work (i.e. to order messages) in parallel during the protocol change, until the old protocol is no longer needed. An important consequence is that the normal flow of messages is not delayed by slower processes.

Even more, the delivery of messages to the user process is neither blocked. Indeed, when the old protocol is finally discarded and uninstalled, the ***Switching protocol*** immediately delivers to the user process the queued messages delivered by the new GCP. After this step, regular delivery with the new protocol is enabled, thus keeping a *normal flow* of messages delivered to the user process.

On the other hand, for this mechanism to properly work, some issues must be considered. These have not been included in the protocol algorithm to simplify its presentation.

First of all, it is needed some way to distinguish the messages broadcast with each GCP. A first solution consists in adding some *header data* in the regular messages but this solution would imply the need of knowing some implementation details, thus making the ***Switching protocol*** dependent on specific GCP implementations.

A second option, general enough to fulfill this requirement is to encapsulate the regular user messages in other messages whose format is only known by the ***Switching protocol***. The protocol can include in these messages additional headers with all the needed meta-data. One of these headers can be used to save an identifier of the GCP used to broadcast the encapsulated user message. From the point of view of the GCPs managed by the ***Switching protocol***, these protocol-dependent messages are as opaque as the regular user messages.

6.3.2 Concurrent Starts

Another issue that can be discussed is the ability of the ***Switching protocol*** to face concurrent starts of the switching procedure. Indeed, in case several protocol switches are started concurrently by different nodes or even the same node, the use of a total order broadcast protocol to broadcast the `PREPARE` messages forces that all the nodes receive the same `PREPARE` messages in the same order.

First of all, multiple `PREPARE` messages can be received by a node. When a `PREPARE` message is received by a node, it starts a new *next_k* iteration, by creating a new P_{next_k} structure. The protocol starts sending messages with the new GCP and queueing in $P_{next\_k}.deliverable$ the messages delivered by it. Each time a new `PREPARE` message is received, a new iteration is started, even if there are some previous GCPs receiving messages.

When the current GCP delivers a message to the ***Switching protocol*** it checks if that message delivery allows to finish the execution of one or more iterations. For this, the `FINISH_PENDING` function is invoked. The only issue to worry about is the proper finalization of the iterations, in the same order they were started. This function checks that, for each iteration started, a corresponding `PREPARE_ACK` message has already been received from all the live processes and all the messages sent by them with the corresponding GCP have also already been received. In this case, the iteration can be considered finished, and the following iteration can be checked.

6.3.3 View Management

When no node failure happens, the behavior of the protocol is that shown in Algorithms 10 and 11.

Nevertheless, the ***Switching protocol*** is able to react to failure notifications provided by an independent membership manager. These are received in the `HANDLE_VIEW_CHANGE` handler. In this handler, we just update the local copy of the set of nodes considered alive and call the `FINISH_PENDING` function. This call is needed because it may happen that the only messages required to finish one or more iterations were sent by processes declared failed. In this call, all the pending iterations are checked, considering only the alive nodes.

The reaction to view changes we present in these algorithms is actually minimum. In Algorithm 12 we extend the initial pseudocode shown in Algorithms 10 and 11. These extensions allow the protocol to provide view change notifications to the upper user process and also manage the join of new nodes. Regarding the first issue, two different alternative guarantees can be provided: *Same View Delivery* and *Sending View Delivery* [33].

If the *Sending View Delivery* property has to be provided, the ***Switching protocol*** has to ensure that all the messages broadcast by the user processes are delivered to them in the view they were sent. In particular, the protocol has to ensure that all the messages broadcast with any of the *pending* GCPs are delivered *before* delivering the following view change notification to the user process. Moreover, once the ***Switching protocol*** learns about a node failure, it has to prevent the user process from sending more messages until the corresponding view change is delivered to it.

For this, we propose the following procedure. When the ***Switching protocol*** is informed about a node failure, it first blocks the sending of user messages. Then, it broadcasts a special `NEW_VIEW` message, with the last GCP started ($P_{next\_k}.GCP$). This message is broadcast with the last GCP started because it is not guaranteed that the previous GCPs are still available in all nodes. The `NEW_VIEW` message includes the set of nodes that compose the new view. After delivering all the pending user messages (those broadcast with any of the started GCPs, including the current one), this `NEW_VIEW` message is eventually delivered to the ***Switching protocol***. The ***Switching protocol*** can then forward the `NEW_VIEW` message to the user process, in order to notify the new view. Finally, it unblocks the sending of user messages.

A few assumptions must be made for this procedure to be correct. First of all, if the *Sending View Delivery* property has to be provided by the *Adaptive Group Communication System*, it must be ensured by the wrapped GCPs. This means that the GCPs must have some kind of *flush mechanism* that must ensure that in case a node fails, before installing the new view, the pending messages are *flushed*. This mechanism may resend and forward messages, if needed, so all the alive nodes receive and deliver the pending messages, although the ***Switching protocol*** does not need to be aware of the details of the *flush* procedures. The second assumption that we need to make is that the `NEW_VIEW` messages broadcast by the ***Switching protocol*** by means of the GCPs, to inform about the view changes are not considered as regular application-level messages by the GCPs but interpreted as membership messages. When the ***Switching protocol*** sends a `NEW_VIEW` message through a GCP, informing about a view change, and especially about node failures, the GCP may start its *flush* protocol. This way we can ensure that the pending user messages are finally delivered.

If the *Sending View Delivery* property is not needed, then the sending of user messages does not need to be blocked. The procedure to follow is thus the same than in the previous case except that the sending of user messages is not blocked. In this case, the user process can go on broadcasting messages after the ***Switching protocol*** receives the node failure notification. Nevertheless, these messages may be delivered to the user process (once totally ordered) after the ***Switching protocol*** delivers the view change to the user process, i.e., in a different view from the one they were sent in, although the total order property provided by all the GCPs ensures that, at least, each message is delivered in the same view to all the user processes. This way, the *Same View Delivery* property is ensured.

The ***Switching protocol*** is also able to manage the join of new nodes. Joins are notified as view changes. In fact, a view change can be viewed as a set of new nodes (nodes that join the system) and a set of nodes that fail.

To implement these features, we propose a number of changes, in Algorithm 12. First we add two new global variables. The *provide_sending_view* variable is a flag used to know if the *Sending View Delivery* property has to be ensured. It's value is set to the value of the *sending* parameter of the `INIT` handler. This way, it can be decided externally. If it is set to $false$, then the *Same View Delivery* property is offered instead. Moreover, we use a *changing_view* global flag, used to know if there is a view change in progress.

The `TO-BCAST` handler is also modified. As a first action, it checks if a view change has been started and if the *Sending View Delivery* property has to be ensured. In this case, the user call to the `TO-BCAST` is blocked. The rest of the handler is the same that the one shown in Algorithm 11.

The `HANDLE_VIEW_CHANGE` handler is also modified. First of all, a new parameter is added, to receive a set of new nodes (i.e., nodes that *join* the system). Then, it broadcasts a special `NEW_VIEW` message, by means of the last GCP started. Finally, the `FINISH_PENDING` function is invoked, as in Algorithm 11.

The `NEW_VIEW` message is received in the new `HANDLE_NEW_VIEW` handler. First, the new nodes are added to the local copy of the set of nodes considered alive. The P data structures from $P_{current\_k}$ to P_{next_k} are updated, to initialize the state corresponding to the new nodes. Then the view change is delivered up to the user process. Finally, in case the *Sending View Delivery* property was required, it unblocks the execution of the `TO-BCAST` handler.

Another issue related to the notification of node failures must be addressed. When a node fails, it may happen that, in several nodes, the corresponding membership service notifies to the ***Switching protocol***, which would broadcast its `NEW_VIEW` message. The result is a number of `NEW_VIEW` messages representing the same node failure are broadcast and received by all nodes. To avoid the multiple notification of a view change to the user processes a simple solution can be adopted.

The ***Switching protocol*** keeps a *view counter* as a global variable. It is initialized to 0 and incremented each time a `NEW_VIEW` is delivered to the ***Switching protocol*** and then forwarded to the user process. Each `NEW_VIEW` message is tagged with the current value of the counter when it is broadcast. If the ***Switching protocol*** receives different `NEW_VIEW` messages with the same value of the *view counter*, it considers the first one and then discards the rest. As the `NEW_VIEW` messages are broadcast in total order, using the last GCP started, all nodes keep the same `NEW_VIEW` message and discard the same other messages.

6.4 Properties of the *Switching protocol*

In this section we provide some properties of the ***Switching protocol*** and some reasoning about their correctness. First, we propose some lemmas used to prove the properties.

Lemma 1: Downwards Validity. *If a user process in a correct node broadcasts a message m, then exactly one of the GCPs of that node eventually broadcasts m exactly once.*

Proof. In the `TO-BCAST` handler, each message sent by the user process is immediately broadcast exactly once, by any of the GCPs currently managed by the ***Switching protocol*** (lines 18–24).

If we consider the modifications presented in Algorithm 12, then, in case the *Sending View Delivery* property is requested and a view change happens, the following message broadcast by the user process may be blocked. In this case, we have to show that the sending is not blocked infinitely.

First, when a view change is notified, then a `NEW_VIEW` message is broadcast (line 118). By the *Validity* property of the GCP used to broadcast the `NEW_VIEW` message, this is eventually delivered by the local node and handled in the HANDLE_NEW_VIEW handler. In this handler, the user process is finally unblocked (line 132) and the message can finally be broadcast, exactly once and using exactly one GCP (lines 107–113). □

Lemma 2: Upwards Validity. *If a GCP delivers a message m to the* ***Switching protocol****, then the* ***Switching protocol*** *eventually delivers m to the user process.*

Proof. It has to be shown that the ***Switching protocol*** does not indefinitely retain a message delivered to it by a GCP. First, if a message m is delivered to the ***Switching protocol*** by $P_{current\_k}.GCP$, then it is immediately delivered to the user process (line 28). If the message is delivered by $P_{k'}.GCP$ (where $current\_k < k' \leq next\_k$), then it is stored in $P_{k'}.deliverable$. In this case, it has to be shown that the message is not retained in that queue infinitely. In other words, it has to be shown that all iterations of the protocol previous to k' are eventually finished.

If m was broadcast with $P_{k'}.GCP$ (with $current\_k < k'$), then we know that a finite number of messages were broadcast with $P_j.GCP$ ($\forall j : current\_k \leq j < k'$). By the *Validity* and *Uniform Agreement* properties of these GCPs, it is known that all those messages are eventually delivered to the ***Switching protocol*** and, by Lemma 1, eventually delivered to the user process. For the same reason, we also know that all the corresponding `PREPARE_ACK` and `PREPARE` messages (used to finish an iteration and start the next one, respectively) are eventually delivered to the ***Switching protocol***. Then, all the iterations previous to $P_{k'}$ are eventually finished. An iteration P_j is finished when all the messages broadcast with the $P_j.GCP$ are delivered to the ***Switching protocol*** (as decided by the `DELIVERY_FINISHED` function). At the end of the iteration P_j, all the pending messages broadcast with $P_{j+1}.GCP$ (those stored in $P_{j+1}.deliverable$) are delivered to the user process (lines 66–75). Then, $current\_k$ is incremented (line 57). Eventually, $current\_k$ reaches k' and message m is finally delivered to the user process. □

Lemma 3: Local Integrity *The* ***Switching protocol*** *delivers a message m to the user process at most once, and only if m has been delivered to the* ***Switching protocol*** *by exactly one of the GCPs of the local node.*

Proof. First of all, the ***Switching protocol*** delivers the message to the user process at most once. If the message is delivered by the current GCP ($P_{current\_k}.GCP$) then, it is directly delivered (line 28). If the message is delivered by a later GCP ($P_{k'}.GCP$, with $current\_k < k'$), then it is first queued (in $P_{k'}.deliverable$). By Lemma 2, we know that the message is eventually delivered to the user process, exactly once (lines 66–75).

On the other hand, it has to be proved that a single message can not be delivered to the ***Switching protocol*** by more than one GCP. Let's suppose that a message is delivered to the ***Switching protocol*** by two different GCPs. The *Uniform Integrity* property offered by these GCPs ensures that they previously sent the message. Nevertheless, this is not possible since the ***Switching protocol*** sends each message only with one of the GCPs (lines 28 and 34). □

Lemma 4: Change Safety *The* ***Switching protocol*** *does not deliver to the user process a message m delivered to the protocol by $P_k.GCP$ after having delivered to the user process a message m' which was delivered to the protocol by $P_{k'}.GCP$, where $k < k'$.*

Proof. If no view change happens, the `TO-BCAST` handler broadcasts the user messages by means of $P_{current\_k}.GCP$ (line 22). As the ***Switching protocol*** does not keep a P_k previous to $P_{current\_k}$, then no message can be broadcast with a previous GCP.

If a GCP change happens, the `TO-BCAST` handler broadcasts the user messages by means of $P_{next\_k}.GCP$ (line 19). The value of *next_k* is incremented each time a GCP change is started (line 41), so $P_{next\_k}.GCP$ is always the last GCP that has been started. If a message is broadcast with $P_{next\_k}.GCP$, then we know that any message subsequently broadcast will be sent with the same GCP or a later one. □

Property 1: Validity. *If a process in a correct node broadcasts a message m, then the* ***Switching protocol*** *eventually delivers m to it.*

Proof. If no GCP change happens, message m is sent with the current GCP ($P_{current\_k}.GCP$). By its $Validity$ property, the GCP eventually delivers m to the ***Switching protocol*** (in the same node). According to Lemma 3 (*Local Integrity*) stated above, the ***Switching protocol*** eventually delivers the message to the user process.

If a GCP change happens, Lemmas 1 (*Downwards Validity*) and 2 (*Upwards Validity*) ensure that the ***Switching protocol*** does not indefinitely retain the *outgoing* messages sent to it by the user process nor the *up-going* messages delivered to it by the GCP. □

Property 2: Uniform Agreement *If the* ***Switching protocol*** *in a node, whether correct or faulty, delivers a message m to the user process, then the* ***Switching protocol*** *in all correct nodes eventually deliver m to their corresponding user processes.*

Proof. Let's suppose that, in one of the nodes, the ***Switching protocol*** delivers a message to the user process. By Lemma 3 (Local Integrity), the message must have been delivered to the ***Switching protocol*** by one of the GCPs. By the *Uniform Agreement* property of the GCPs, in all the correct nodes, the GCP delivers the message to the ***Switching protocol*** and by Lemma 2 (Upwards Validity), the ***Switching protocol*** eventually delivers up the message to the user process in all correct nodes.

In case the GCPs do not satisfy the *Uniform Agreement* property but just a *Non-uniform Agreement* property, then the property satisfied by the ***Switching protocol*** is not *Uniform Agreement* but just the corresponding *Non-uniform Agreement* property. □

Property 3: Uniform Integrity *For any message m, the* ***Switching protocol*** *of every node, whether correct or faulty, delivers m at most once to the user process and only if m was previously broadcast by its sender.*

Proof. First of all, it has to be shown that a user process does not deliver a message twice.

First, by Lemma 3, we know that the ***Switching protocol*** can not deliver twice the same message. It delivers a message twice only if the GCP has delivered twice that message to it.

By the *Uniform Integrity* property of the GCP, this can only happen if the GCP in the sender node has broadcast twice the message. By Lemma 1 (*Downwards Validity*), we know that this is only possible if the sender node broadcasts twice the same message through the GCP, and this can only happen if the user process in the sender node broadcasts twice the same message.

Moreover, it has to be shown that the ***Switching protocol*** only delivers a message to the user process if the message was previously broadcast by its sender node.

First, it is known that the ***Switching protocol*** only delivers to the user process messages that have previously been delivered to it by one of the GCPs (lines 28). By the *Uniform Integrity* of the GCPs, this only happens after the GCP in the sender node has broadcast the message. The ***Switching protocol*** itself ensures that this can only happen after it has broadcast the message through the corresponding GCP in the sender node. □

Property 4: Uniform Total Order *If the* ***Switching protocol*** *in any nodes p and q, whether correct or faulty, both deliver messages m and m', then the* ***Switching protocol*** *in p delivers m to its user process before m' if and only if the* ***Switching protocol*** *in q delivers m to its user process before m'.*

Proof. Let's suppose that the ***Switching protocol*** in both nodes p and q delivers two messages m and m'. If p delivers both m and m' using the same GCP, by the *Uniform Total Order* property of the GCP and by protocol construction, it is known that all the nodes will deliver m and m' in the same order, using the same GCP.

Now let's suppose that p delivers m using $P_k.GCP$ and delivers m' using $P_{k'}.GCP$, with $k < k'$. Then, q also delivers m using $P_k.GCP$ and m' using $P_{k'}.GCP$. Moreover, by Lemma 4 (*Change Safety*), as m has been broadcast using $P_k.GCP$, q delivers m to the user process before delivering any other message broadcast by $P_{k'}.GCP$, which means that q delivers m prior to m'.

The reasoning is also valid if p or q fail after delivering m and m', respectively. On one hand, p and q deliver m and m', as long as $P_k.GCP$ and $P_{k'}.GCP$ satisfy the *Uniform Total Order* property. On the other hand, by Lemma 4 (*Change Safety*), both nodes deliver all the messages broadcast by $P_k.GCP$ before starting to deliver messages broadcast by $P_{k'}.GCP$. As a result, both p and q deliver m before delivering m'. $\square$

6.5 Experimental Evaluation of the *Switching protocol*

In this section we describe an experimental evaluation of the ***Switching protocol*** we performed in order to provide a *proof of concept* of its operation. First, we describe the environment and the methodology used to perform the evaluation. Then, we provide some results to show the effectiveness of the protocol.

6.5.1 Introduction

To test the ***Switching protocol***, we have implemented a simple Java test application, to be ran in a number of nodes. The overall architecture of the system running in each node is shown in Figure 6.2, which is a simplified version of that depicted in Figure 6.1. The application acts as a *client* of the ***Switching protocol*** which in turn wraps several of the total order protocols we implemented to perform the experimental evaluations.

The message transport layer used in this experimental evaluation is a new transport layer we implemented on top of the JBoss Netty 3.2.4 networking library [4]. Netty is a client/server library that implements the Java NIO specification [5] and offers asynchronous event-driven abstractions for using I/O resources. In our system, Netty allowed us to build a reliable, high performance, stream oriented, TCP-like message transport layer used by the group communication protocols to unicast and broadcast messages. The reason why we decided to switch to Netty is because its non-blocking architecture allows the system in general, and the group communication protocols in particular, to perform better.

On the other hand, the implemented system does not include a *Membership service* since we are not considering view changes in this evaluation. Moreover,

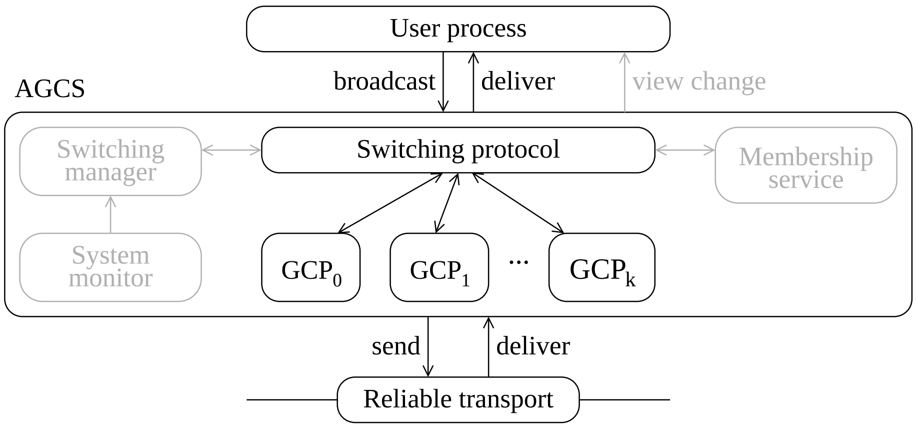


Figure 6.2: Architecture of a node (simplified version)

the implemented system does not include the *System monitor* or the *Switching manager* shown in the original figure. Instead, the test application itself is in charge of issuing the *start-of-change* events to request a GCP change, to any of the available *pre-loaded* GCPs (***UB***, ***TR***, etc.), as described later.

6.5.2 Environment

The environment is very similar to the one used in the previous experimental evaluations. The application is executed in a system composed of four nodes. Each node is a different physical machine with an Intel Pentium D 925 processor running at 3.0 GHz and 2 GB of RAM, running Debian GNU/Linux 4.0 and Sun JDK 1.5.0. The nodes are connected by means of a 24-port 100/1000 Mbps DLINK DGS-1224T switch. As in the previous experimental evaluations, the switch keeps the nodes isolated from any other node, so no other network traffic can influence the results.

6.5.3 Test Application and Methodology

The test application is a regular Java console application that is run in each of the four nodes of the system. In each node, the application broadcasts to all nodes a sequence of messages by handling them to the ***Switching protocol*** as if it were a regular total order protocol.

As in Section 5.2, the messages are broadcast at a uniform sending rate, configured externally. We have performed a number of tests with different sending rates. As in Section 5.2, no other message flow control mechanism has been used. Moreover, messages are also tagged with random priorities in a similar way and the length of the messages is also variable but they still fit into one wire-level packet.

To perform the evaluation of the ***Switching protocol***, we have run the test application under different configurations. In a first set of executions, we configured the ***Switching protocol*** to use the ***UB*** (sequencer-based) and the ***TR*** (privilege-based) regular (non-prioritized) protocols. The application was configured to periodically request a GCP change each 5000 ms. Thus, the ***Switching protocol*** starts using the ***TR*** protocol and after 5000 ms the ***Switching protocol*** is asked to *switch* to ***UB***. After the next 5000 ms, the application asks to change to ***TR*** and so on.

In each test, the application was configured to broadcast messages at a fixed sending rate. We ran different tests using rates of 40, 60, 80, 120 and 130 messages broadcast per second and node. Thus, the global sending rates range from 160 to 520 messages per second.

In each execution we measure the *delivery time* of the messages, computed as the time observed by the application in a given node, from the moment in which it broadcasts the message to the moment in which it receives back the message, once totally ordered, exactly in the same way as we did in Section 5.2. This means that for each node, we get a series of delivery times corresponding to the series of messages broadcast by that node.

Moreover, in order to know about the *distribution in time* of the message deliveries, we also count the number of messages that are delivered in each *hundredth* of a second. This numbers allow us to know if there is a regular *flow* of messages being delivered.

This set of experiments is repeated, using the ***UB_PRIO*** and ***TR_PRIO*** protocols.

6.5.4 Results

In Figures 6.3, 6.4, 6.5, 6.6 and 6.7 we show the *delivery times* recorded by a single node of the system in the *first* set of experiments, with ***TR*** and ***UB*** and a sending rate of 40, 60, 80, 120 and 130 messages broadcast per second and node, respectively. In Figures 6.8, 6.9, 6.10, 6.11 and 6.12 we show the corresponding count of messages delivered in each hundredth of a second, using a sending rate of 40, 60, 80, 120 and 130 messages broadcast per second and node, respectively.

In Figures 6.13, 6.14, 6.15, 6.16 and 6.17 we show the *delivery times* recorded by a single node of the system in the *second* set of experiments, with ***TR_PRIO*** and ***UB_PRIO*** and a sending rate of 40, 60, 80, 120 and 130 messages broadcast per second and node, respectively. In Figures 6.18, 6.19, 6.20, 6.21 and 6.22 we show the corresponding count of messages delivered in each hundredth of a second, using a sending rate of 40, 60, 80, 120 and 130 messages broadcast per second and node, respectively.

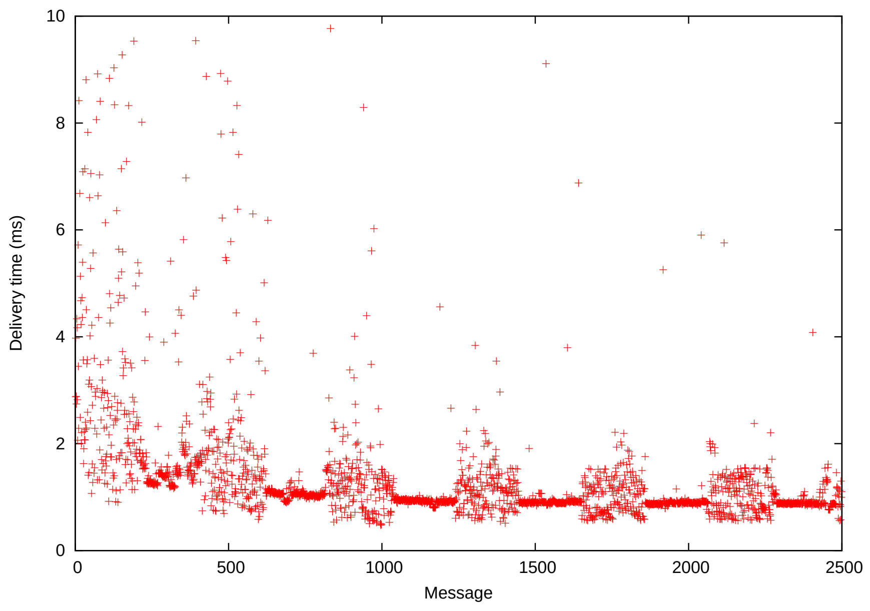


Figure 6.3: Delivery times (40 msg/s with ***TR*** and ***UB***)

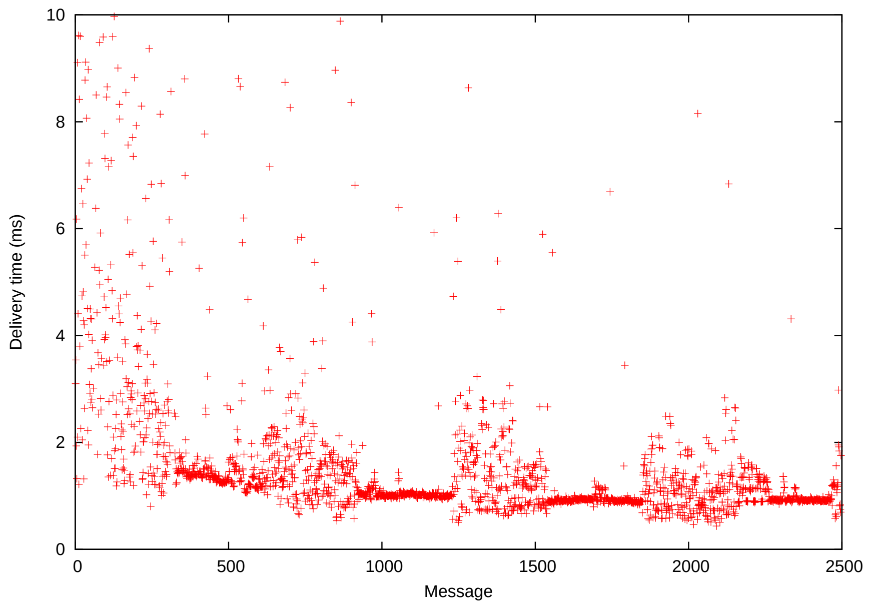


Figure 6.4: Delivery times (60 msg/s with ***TR*** and ***UB***)

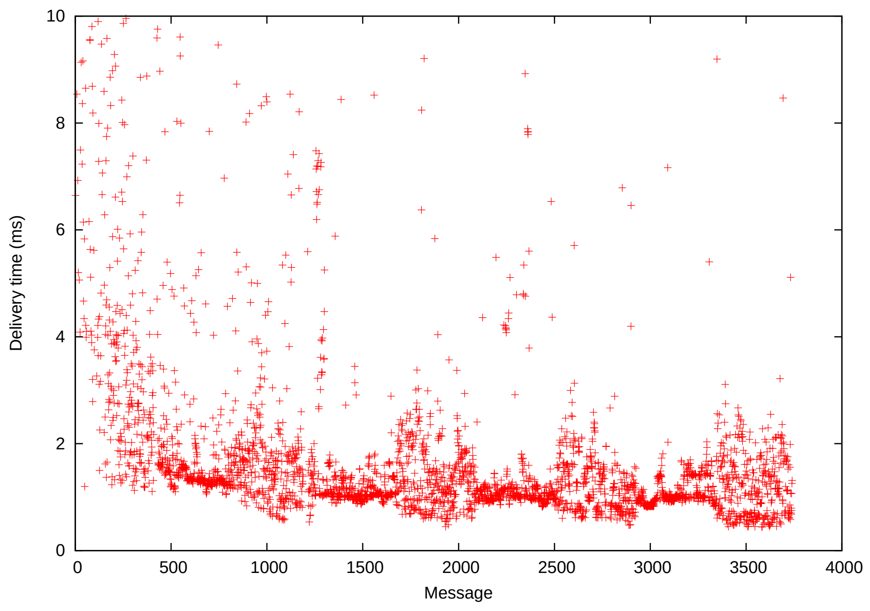


Figure 6.5: Delivery times (80 msg/s with ***TR*** and ***UB***)

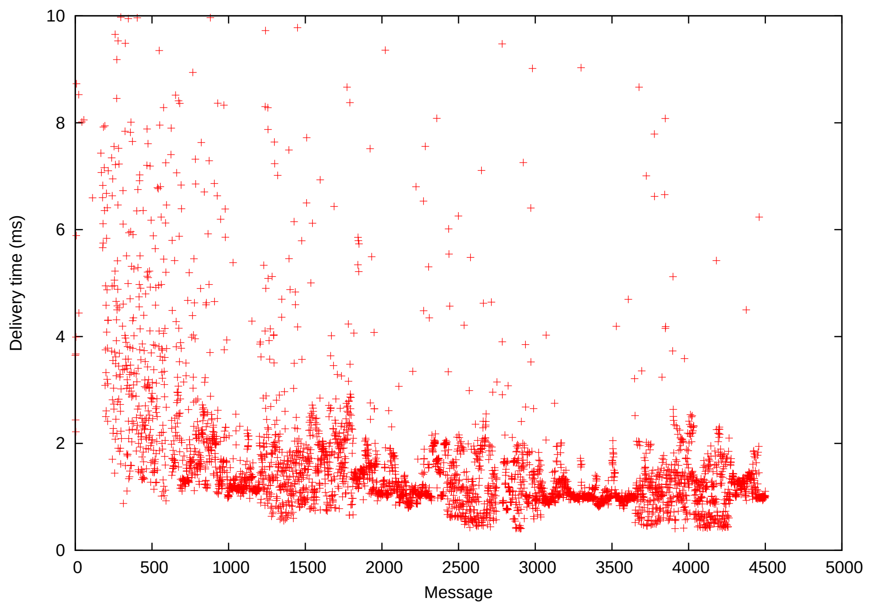


Figure 6.6: Delivery times (120 msg/s with ***TR*** and ***UB***)

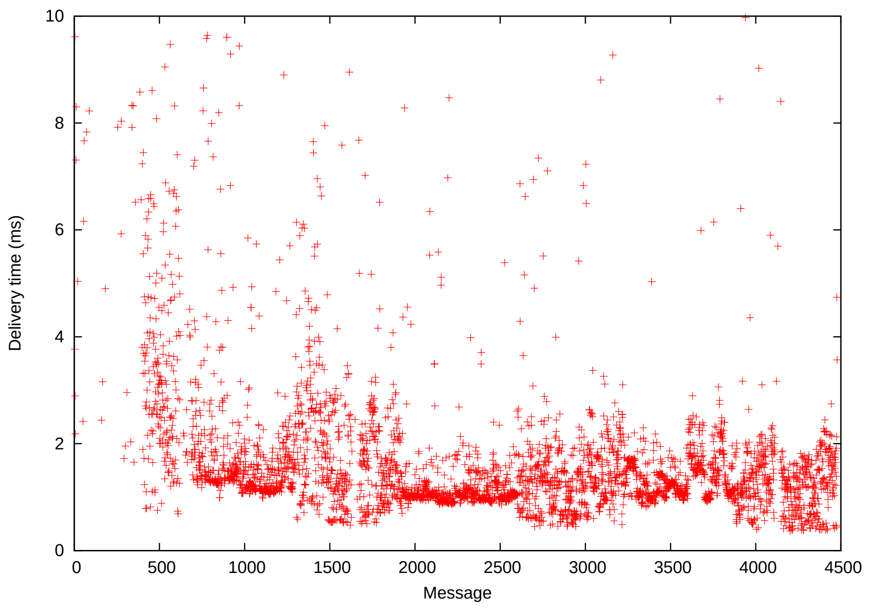


Figure 6.7: Delivery times (130 msg/s with ***TR*** and ***UB***)

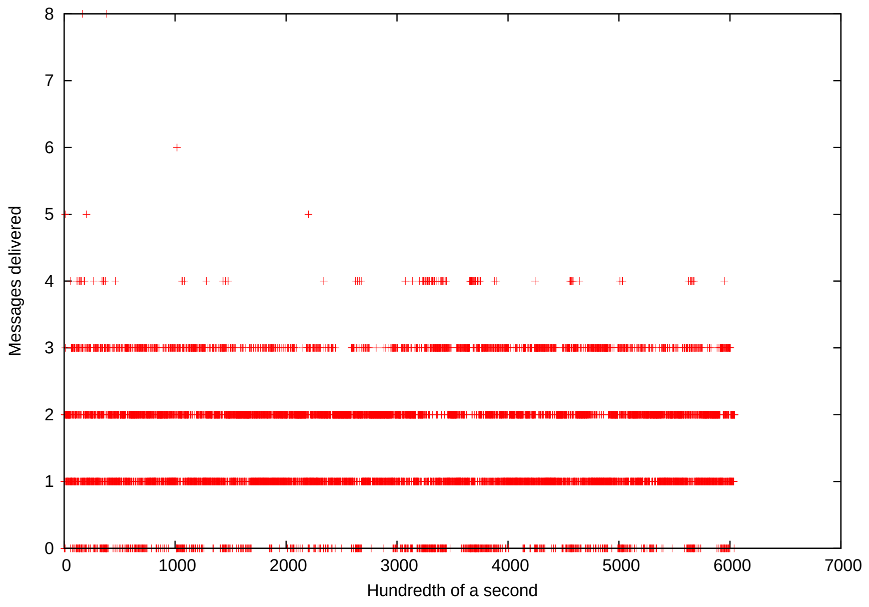


Figure 6.8: Messages delivered by hundredth (40 msg/s with ***TR*** and ***UB***)

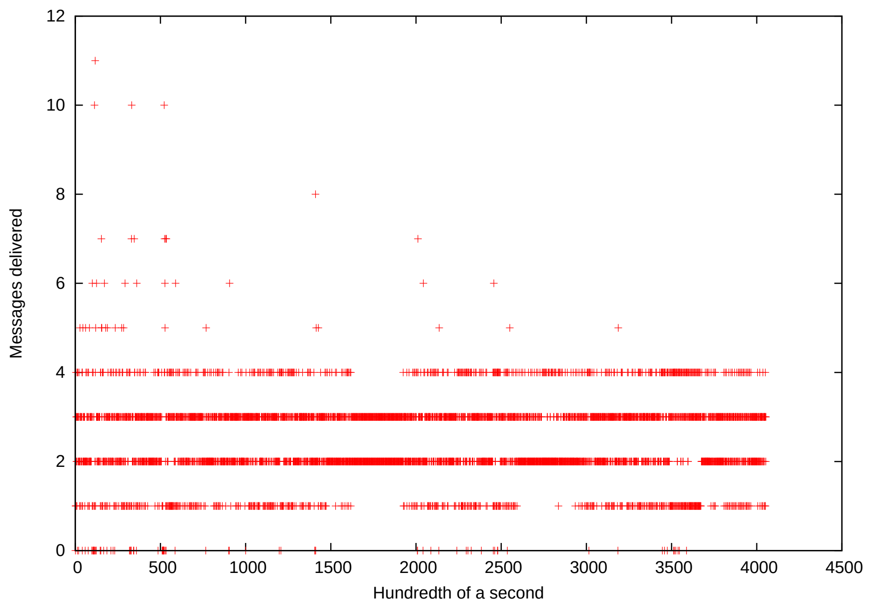


Figure 6.9: Messages delivered by hundredth (60 msg/s with ***TR*** and ***UB***)

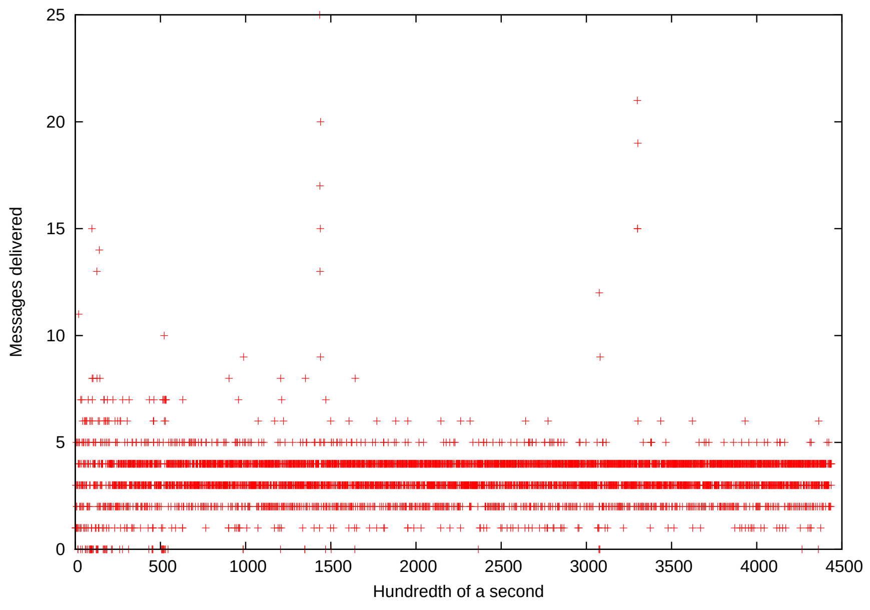


Figure 6.10: Messages delivered by hundredth (80 msg/s with ***TR*** and ***UB***)

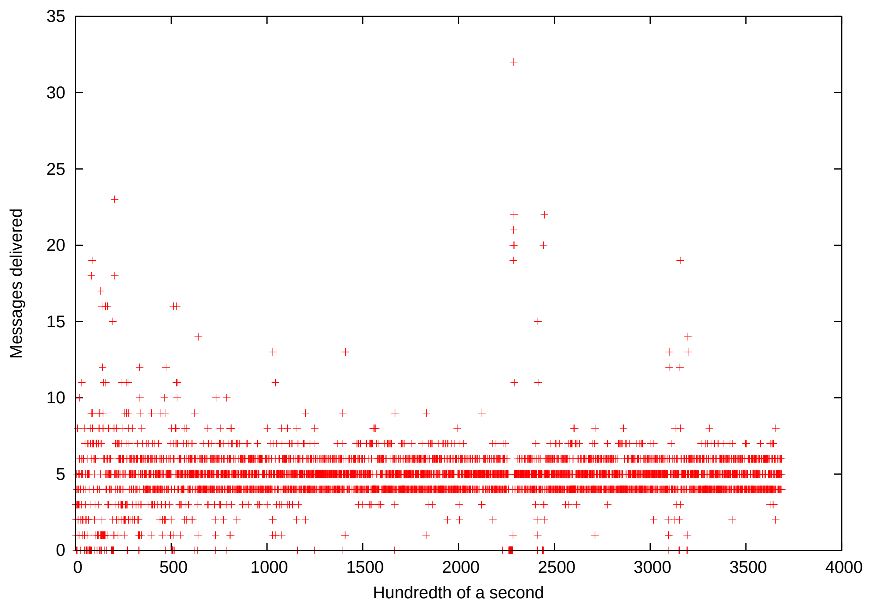


Figure 6.11: Messages delivered by hundredth (120 msg/s with ***TR*** and ***UB***)

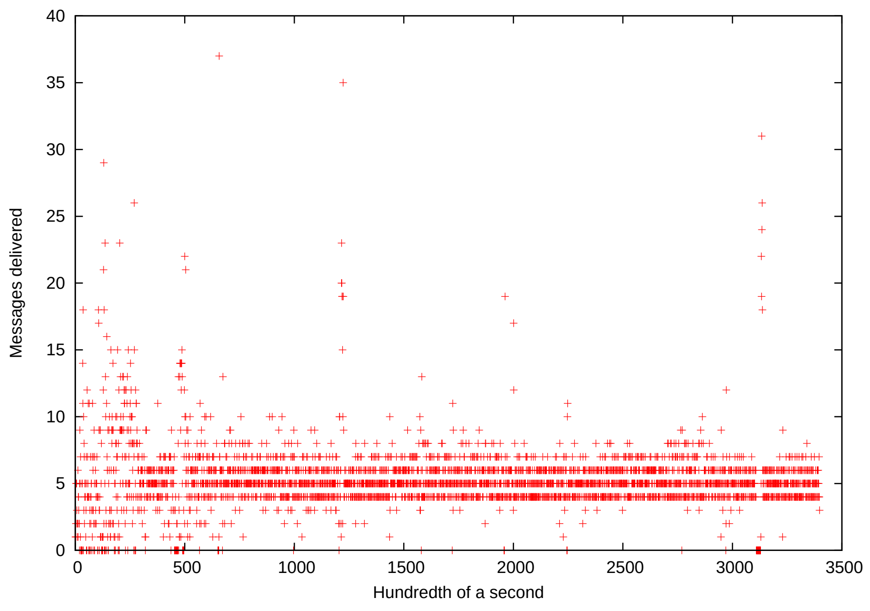


Figure 6.12: Messages delivered by hundredth (130 msg/s with ***TR*** and ***UB***)

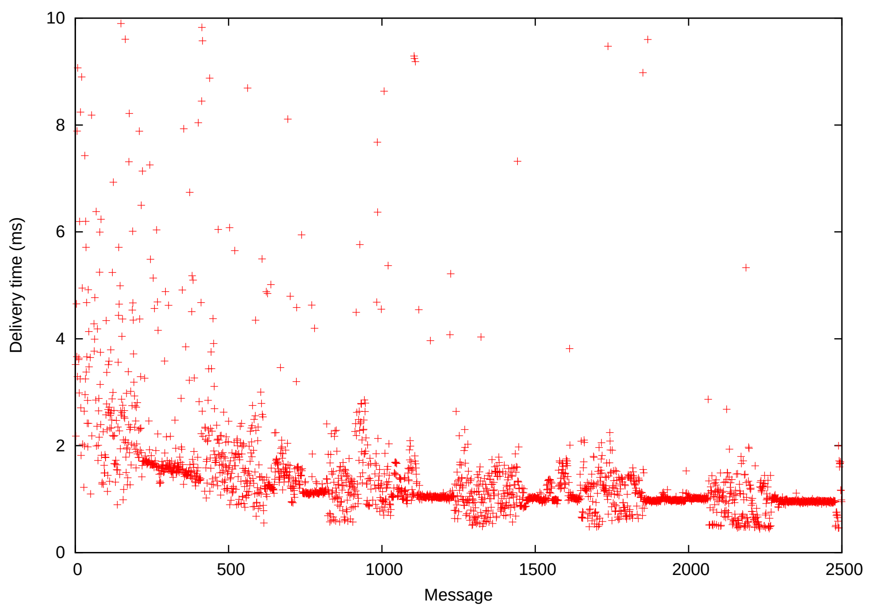


Figure 6.13: Delivery times (40 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

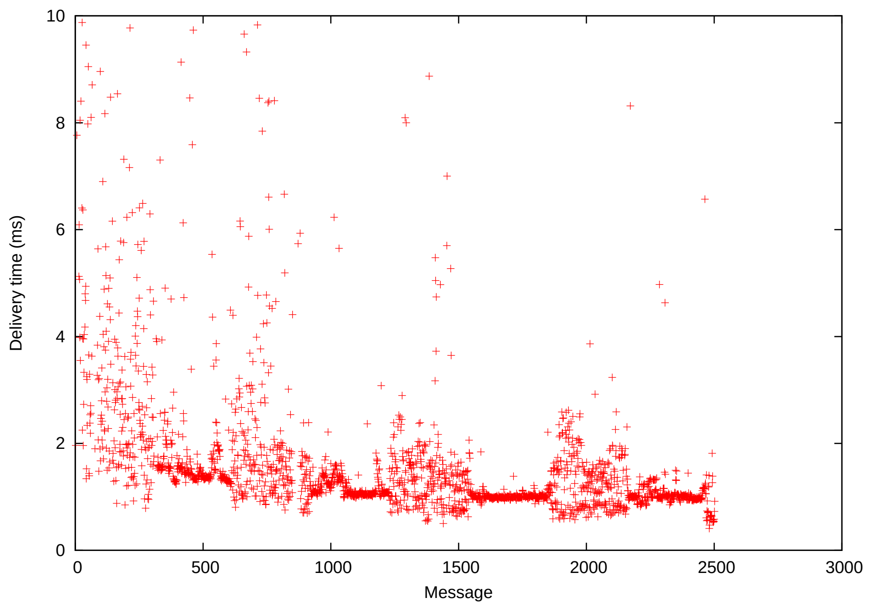


Figure 6.14: Delivery times (60 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

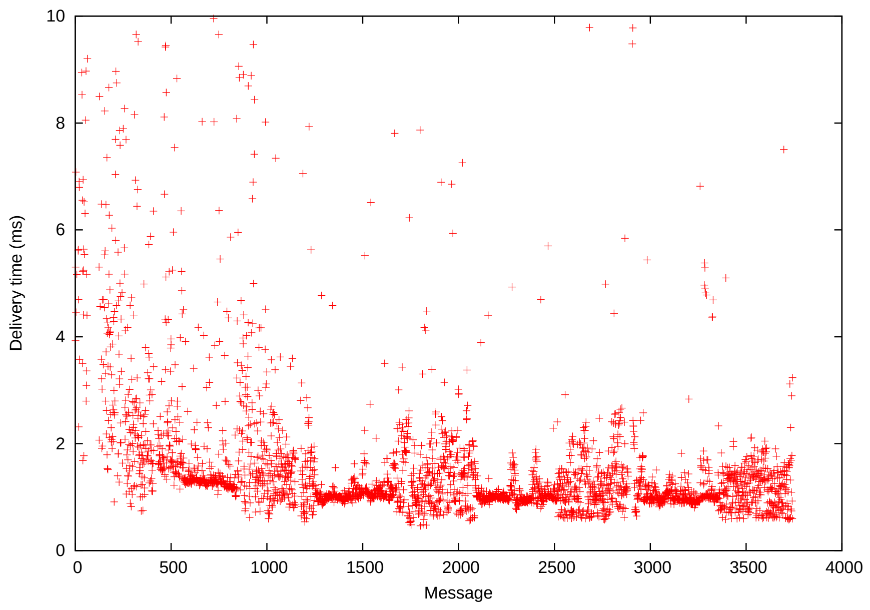


Figure 6.15: Delivery times (80 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

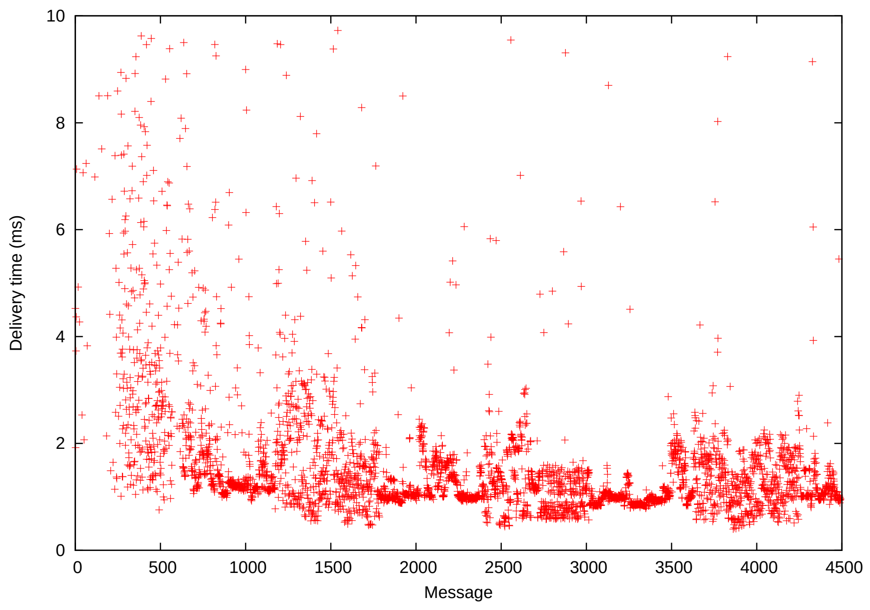


Figure 6.16: Delivery times (120 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

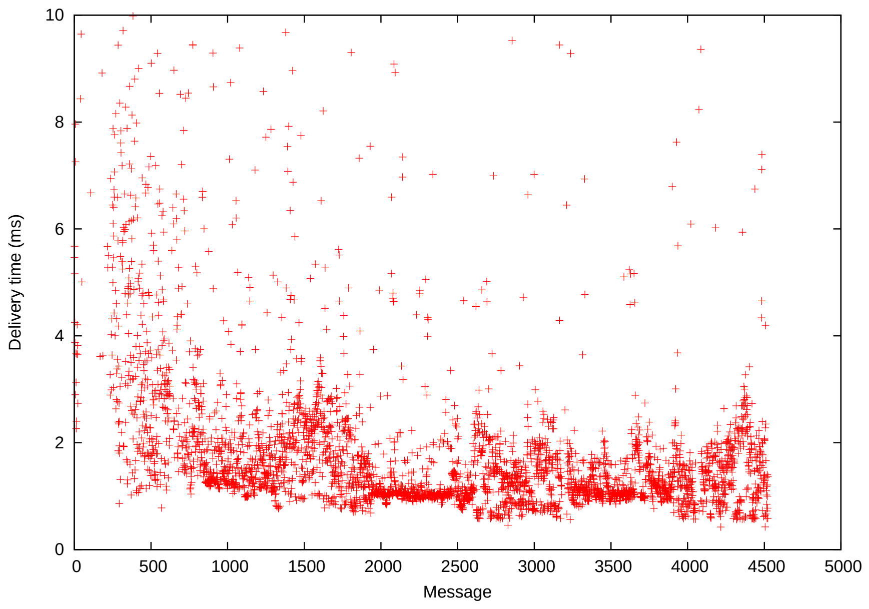


Figure 6.17: Delivery times (130 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

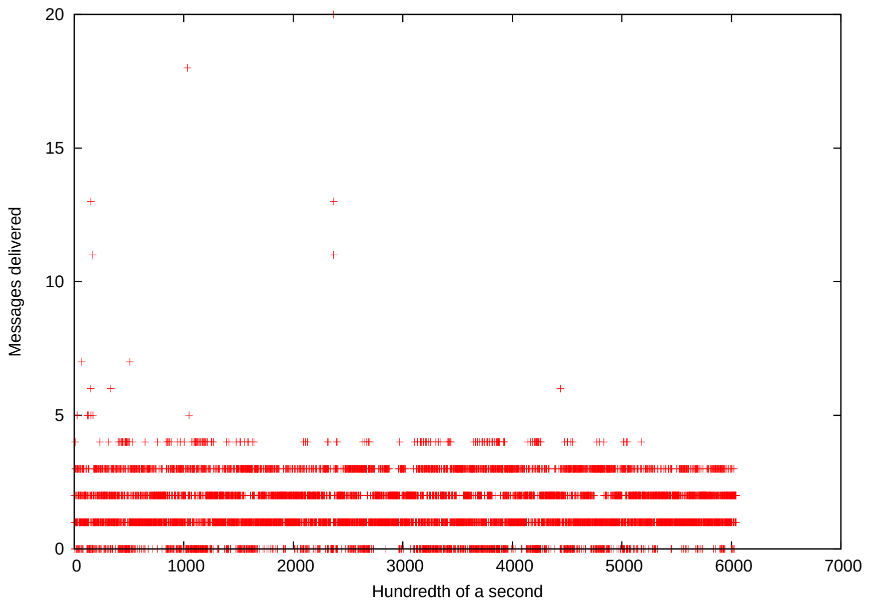


Figure 6.18: Messages delivered by hundredth (40 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

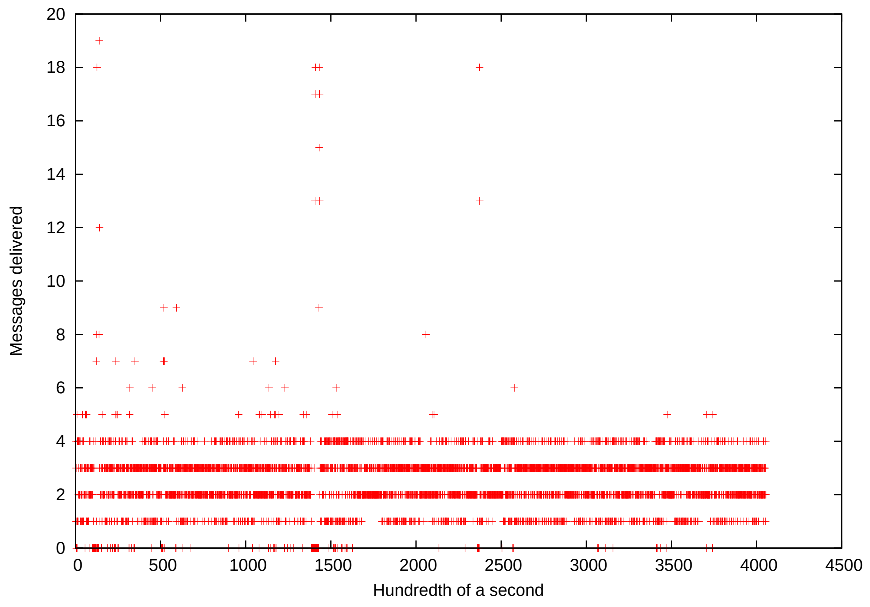


Figure 6.19: Messages delivered by hundredth (60 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

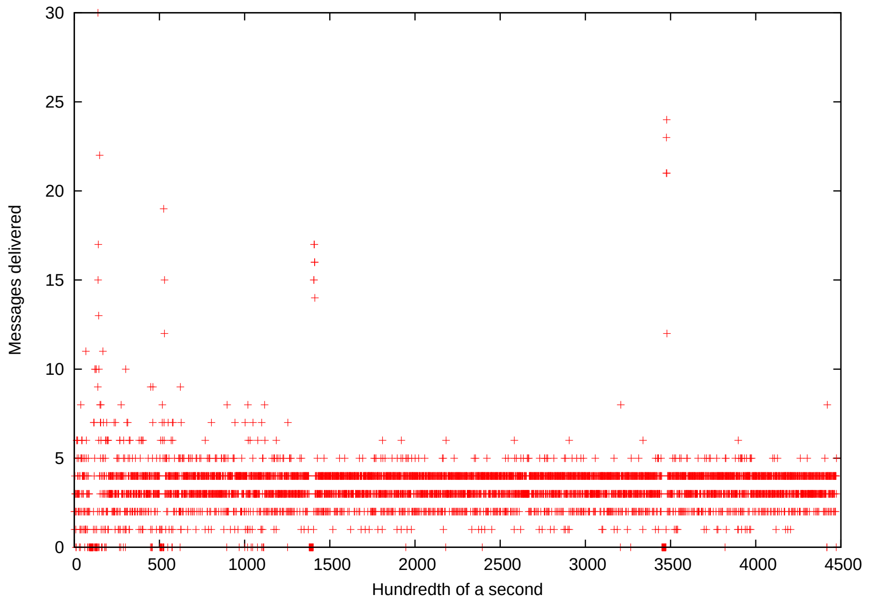


Figure 6.20: Messages delivered by hundredth (80 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

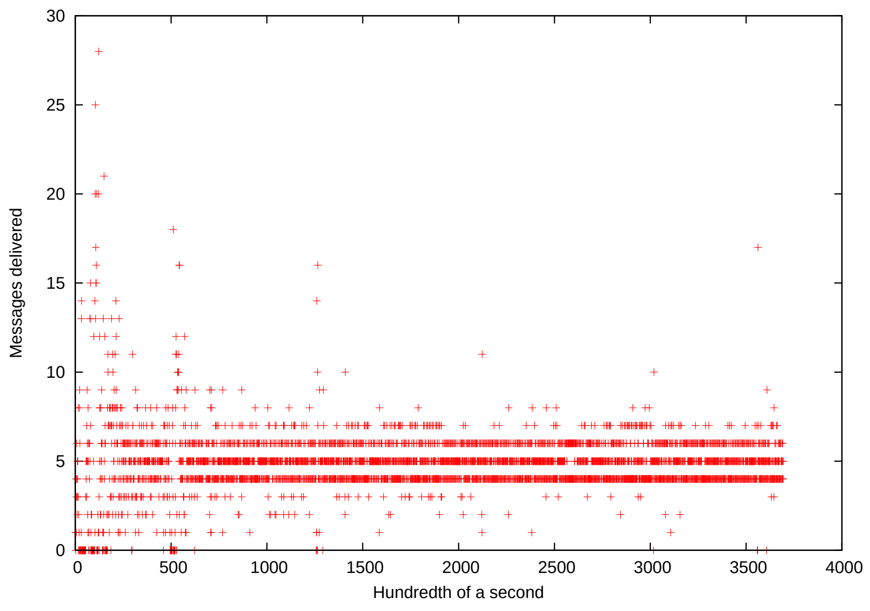


Figure 6.21: Messages delivered by hundredth (120 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

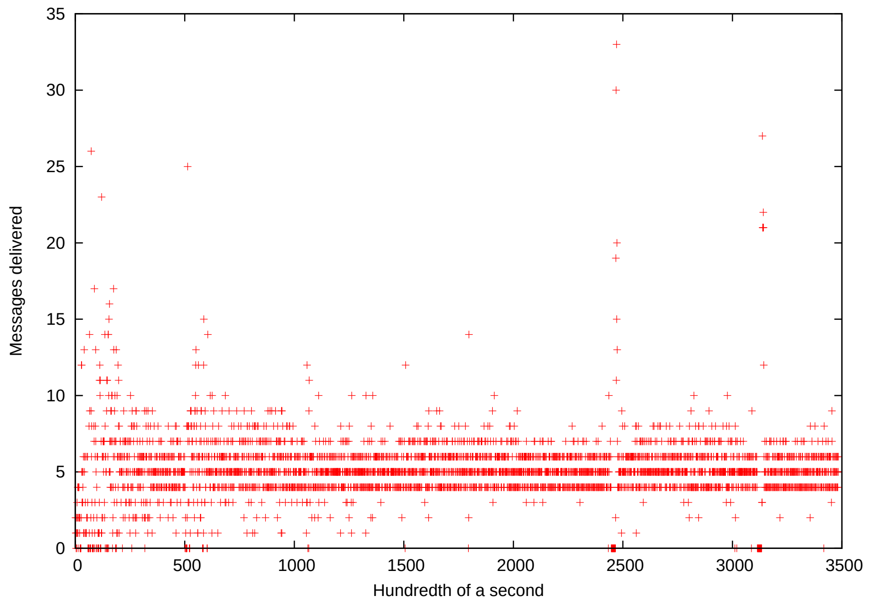


Figure 6.22: Messages delivered by hundredth (130 msg/s with ***TR_PRIO*** and ***UB_PRIO***)

6.5.5 Discussion

In Figures 6.3 to 6.7 we show the delivery time of the messages. After each test we get a series of messages, which have been delivered in total order. In a given series, the i_{th} message is represented by $x = i$ and its delivery time (in milliseconds) is $y(i)$. As explained in Section 6.5.3, in these tests the ***Switching protocol*** switches between two total order protocols (e.g. ***TR*** and ***UB***). This means that in any test, the application delivers a *sub-series* of the messages with the first protocol (in Figure 6.3, about 200 messages with ***TR***), then it delivers another *sub-series* with the second protocol (in Figure 6.3, another 200 messages, with ***UB***) and so on.

These figures show that the ***Switching protocol*** does not increase the delivery time of the messages.

In these figures we can notice two different *distributions* of the message delivery times. The delivery times of the messages delivered with ***TR*** show a higher *variability* than the times corresponding to messages delivered with ***UB***. The reason of this behavior is because according to the ***TR*** protocol, a node needs to have the *privilege* to send messages. In ***TR*** and ***TR_PRIO***, this is implemented by means of a rotating *token message*. To broadcast a (totally ordered) message, the nodes typically have to wait some *variable time*, until they get the token. This extra delay is the main responsible of the higher *variability* observed in Figures 6.3 to 6.7.

As the sending rate is increased, the delivery times got with ***UB*** tend to increase because the sequencer node is more and more *busy* sequencing messages. The time increment becomes more and more variable, thus increasing the variability of the final message delivery times. Nevertheless, these differences are not actually so important, since even in these cases, the figures show that the ***Switching protocol*** is not introducing any significant delay in the message delivery times. This can be checked by analyzing the delivery times in each *sub-series*. In case that the ***Switching protocol*** introduced a delay in the delivery times, this delay would have been noticeable. Specifically, the delivery times at the beginning of each *sub-series* would have been noticeably higher than the delivery times of the rest of the *sub-series*. As the figures show, the delivery times in a given series are quite similar and comparable among them (apart from several punctual times that can be considered *anomalous*). From this behavior, we can draw the conclusion that the ***Switching protocol*** is not introducing any significant delay in the message delivery times.

On the other hand, those figures allow us to assess the cost of delivering each message but they do not provide any information about how that message delivery is being *distributed* over time. Figures 6.8 to 6.12 show the number of messages delivered every hundredth of a second (i. e. the interval [0:100] corresponds to one second). These figures allow us to know about the message delivery over time.

For instance, in Figure 6.8 we can see that in most of the *one hundredth of a sec-*

ond intervals, the ***Switching protocol*** is delivering between 1 and 3 messages. As the sending rate is increased (Figures 6.9 to 6.12), this delivery rate also increases. For instance, in Figure 6.12, between 5 and 6 messages are delivered in *one hundredth* interval.

The importance of these figures is that in all cases, the number of messages delivered by hundredth of a second follow a *quite regular distribution*, in spite of the successive protocol changes that have happened. These figures also show a small number of anomalous values but it can be seen that they do not happen at instants of time which are multiple of 500 hundredths of second (5000 ms) but at any time, which means that they are not directly caused by a protocol switch, but by some other reason (for instance, due to the thread scheduling policies of the operating system or the Java Virtual Machine). Thus, we can assert that the number of messages delivered per time unit does not depend on whether a protocol switching is being carried on and for this reason, we can finally conclude that the ***Switching protocol*** is not producing interruptions or delays in the *flow* of message deliveries.

Figures 6.13 to 6.22 show the corresponding results when using the ***TR_PRIO*** and ***UB_PRIO*** protocols. These results are very similar to those depicted in Figures 6.3 to 6.12, which allows us to conclude that the ***Switching protocol*** is working properly when switches among any kind of total order protocols (prioritized or not): it does not impose significant time overheads in the message delivery times and it does not interrupt or delay the message delivery.

This shows that the ***Switching protocol*** is useful to adapt an application to changing requirements and load conditions. The figures we have presented show that some total order protocols are able to minimize the dispersion of the message delivery times while others lead to a reduction in the mean delivery time of the messages. The proposed switching support we have proposed allows the applications to switch among different total order protocols under changing conditions, without suffering significant performance penalties during the protocol switch.

6.6 Related Work

In this section we briefly review some previous work that is related to our concern. The reviewed papers are divided into two different groups. In a first group, we include those papers that propose some configurable architecture or mechanism that is able to adapt to changing environments or settings, by means of *tuning* its behavior, but without performing structural changes like the ones carried on by a switching protocol. We also include some other work directly related to adaptable systems. In a second group, we include those papers that use some *dynamic switching* mechanism that is able to replace the current implementation of one or several services. The papers in both groups are presented chronologically. Some of the references cited present a solution based on a switching algorithm while others present work related to adaptive systems from a more general point of view.

6.6.1 Configurable Systems

Composability in x-kernel (Hutchinson et al., 1991) and Coyote (Bhatti et al., 1995)

The x-kernel operating system kernel [51] was designed to ease the design and development of network protocols. It is considered one of the first systems based on *composable* stacks of protocols. For instance, it includes protocols that implement different communication standards like IP, UDP, TCP and even low level protocols like ARP.

In x-kernel, the composition of the protocol stacks to use is defined statically, in configuration time. In boot time, each protocol communicates with its underlying protocol in order to agree on the relationship. Once the kernel is booted, there is no mechanism to dynamically change the composition of the protocol stacks used by x-kernel.

The Coyote system [23] is based on x-kernel and proposed the decomposition of regular x-kernel protocols into a set of *micro-protocols*. In configuration time, a first construction step if performed, by combining the micro-protocols of a protocol. Then, a regular x-kernel configuration step is carried, to build regular x-kernel protocol stacks.

As in x-kernel, the configuration of Coyote is static and no dynamic reconfiguration mechanism is available.

ADAPTIVE (Schmidt, 1993)

The ADAPTIVE system [93] is an environment to develop network protocols, designed to adapt to heterogeneous and changing environments. First of all, it offers a number of high-level abstractions to specify the behavior of the network services that will be finally offered to the user application, according to the current setting (e.g. the topology and type of the network) and the application quality-of-service needs. The specifications of the services are used to instantiate *protocol machines*, which are protocol implementations available in a repository and tuned to fit such requirements and needs. Moreover, in configuration time, user applications can refine the specification of the network protocols to use. The reconfiguration mechanism is then able to tune the current protocol machines or create new ones, in order to adapt to the application needs.

About the Use of Standard Interfaces (Wiesmann et al., 2003)

Several efforts have been made to propose a set of standard interfaces that express a wide range of group communication services like group membership, or communication primitives.

In [103], the authors propose the use of middleware architectures built up from components that follow standard and well-known interfaces. The architecture they propose can be used to build distributed systems and it includes a membership service, a fault detector service and some messaging services. These services are implemented by components that must offer well-defined and standard interfaces to the components *above them* (i.e., components that *use* the services they offer). These components, in turn, use the services offered by the components *below* them.

The benefit of using standard interfaces is twofold. First, as the knowledge of the standards to use can be reused, the design and implementation of new components is easier and simpler. Moreover, the use of standard interfaces allows the replacement of the implementation of a given component by a new one, as justified in previous sections.

In [103], for each service, several standard alternatives are considered. For instance, TCP/IP UDP/IP, IP-multicast, BEEP, APEX and JMS are considered as standards to define the behavior of the messaging components while LDAP and SNMP standards are considered for the membership service and SNMP and CMIP, for the fault detection service.

A Survey of Middleware Software (Sadjadi, 2003)

In [89], a survey of configurable and adaptive middleware is presented. This work is actually a first version of [68], which is reviewed in a later section. In the survey, a number of solutions are classified into different classes.

The survey first identifies four key technologies that offer *composability* and *adaptability*: a) computational reflection [66], based on the use of introspection, b) component-based design, c) aspect-oriented programming [60] and d) software design patterns.

A first classification in [92] classifies middleware software depending on the abstraction layer in which they may be placed: a) *Host-infrastructure*, b) *distribution*, c) *common-services* and d) *domain-services*.

A second classification proposed in [89] classified middleware software according to its adaptation level: a) *configurable*, b) *customizable*, c) *tunable*, and d) *mutable*. The *mutable* class is the only one that can be considered completely dynamic and may include some techniques like introspection, aspect-based programming and dynamic code loading.

Finally, in [89] a third classification is proposed that classifies middleware according to its application domain: a) *QoS-oriented systems*, b) *dependable systems*, and c) *embedded systems*.

In the survey a big number of solutions are reviewed, many of them related to CORBA. Nevertheless, none of them can be compared to the switching mechanism proposed in Section 6.2 or the other solutions reviewed in this section.

A Taxonomy of Compositional Adaptation (McKinley et al., 2004)

In [68] (an extended version of [67]), a survey of adaptive systems is presented.

In this survey, two main types of adaptation are identified. *Parameter adaptation* is present in systems that are able to modify the values of their parameters and variables in order to adapt to changes in their settings, environments, working conditions, etc. On the other hand, *compositional adaptation* involves the ability to algorithmically or structurally change a system in order to perform such adaptation. In the survey, a *taxonomy of compositional adaptation* is presented.

The taxonomy is multidimensional. The solutions surveyed are classified according to three different criteria: a) how, b) when and c) where to *compose* (i.e. perform a system's adaptation).

Regarding to *how to compose*, several mechanisms can be used.

- Redirection of function pointers. The pointers that point to the functions that contain the code to change or *adapt* can be *redirected* to point to different functions (for instance, *proxy* functions).
- *Wrappers.* The use of the *wrapper pattern* allows business objects to be encapsulated by *wrapper objects* that can control the original objects.
- *Proxies.* According to the *proxy pattern*, some *proxy* code can be *inserted* in the original code, to intercept and manage regular invocations to business logic.
- The *strategy pattern.* Each service implementation is encapsulated under an interface. This allows the replacement of a given implementation by another one, as long as both share the same interface.
- *Virtual components.* A *virtual component* is a placeholder that allows the loading and unloading of service code in an application-transparent manner.
- *Meta-Object Protocols.* A specific protocol can be used to dynamically replace the implementation of a service.
- *Aspect weaving.* Aspect Oriented Programming can be used to *inject* orthogonal functionality to existing service implementations.

- Middleware interception. Regular service requests and the corresponding responses are intercepted at a middleware-level layer. Adaptation can be performed in such layer.
- Integrated middleware. Besides indirectly using a middleware layer, the user applications can also explicitly make use of their services.

Additional subcriteria are considered regarding this criterion: transparency of the solution, granularity, coverage and support of standards.

The *transparency* criterion expresses the transparency level of the adaptation mechanism respect to the functional code of the application, the adaptive code, the distribution middleware services (if any) and the virtual machine (if any). The *granularity* criterion is useful to know the granularity of the adaptation mechanism (per system, per class, per object, per method or per invocation). The *coverage* criterion distinguishes systems that only are appliable to local invocations from those that also consider remote invocations. Moreover, the ability to apply the adaptive mechanism to just a subset of the invocations is also checked. Finally, the *standards support* criterion allows to know which standard like CORBA/CCM, Java RMI/J2EE and DCOM/.NET are supported by the systems.

Regarding to *when to compose*, two first categories can be distinguished: *static composition* and *dynamic composition*. Static composition is performed in configuration, compilation, deployment, linking or even loading time while dynamic composition, is performed in run-time. Static composition is usually easier to perform but it is usually less flexible and powerful than dynamic composition, which is, on the other hand usually more complex to perform.

As there are different levels of static composition, it can be achieved in different manners. Simpler static composition can be performed by tuning hardwired parameters and code and recompiling the system. More flexible mechanisms perform the adaptation in deploy time, by choosing the proper components and modules to use. The most powerful alternatives include late binding and dynamic class loading.

On the other hand, systems that use dynamic composition can be *tunable* or *mutable*. *Tunable software* can be dynamically configured and *adapted* by run-time tuning some of their parameters and variables. *Mutable software* offers the possibility of altering the functionality of the system, for instance, by dynamically replacing the code of some of their components.

Regarding to *where to compose*, middleware-level and application-level alternatives can be considered. Middleware-level alternatives include constructing a layer of adaptable software and attach it to the user application and modifying a virtual machine in order to add some adaptive support. Application-level alternatives imply adapting part of the application itself. Different alternatives exist like the use of programming languages that natively offer some *adaptation support* (like CLOS or Python), the extension of the programming language run-time mechanism used by the application or the use of Aspect Oriented Programming libraries.

In the survey, more than forty solutions are classified according to this taxonomy, including classic Group Communication Systems (Ensemble, Totem and others) and CORBA middlewares (ACE, TAO, CIAO and others).

A Standard GCS Interface (GORDA project, 2007)

In [29], another proposal is presented. The idea is to have a middleware layer that provides an abstract Application Program Interface to be used by conventional distributed systems. This layer is placed between an application and a Group Communication System, thus acting as an adapter of the latter.

This strategy yields two major benefits. First, it avoids the use of implementation-specific semantics and interfaces. Moreover, it isolates applications from a specific GCS implementation and thus allowing a future replacement of the current implementation. As a side effect, this independence also eases the evaluation of the behavior and performance of an application using different GCS implementations, for instance, in order to choose one of them.

This strategy is implemented in the *Group Communication Service* project [35]. Nowadays, this project includes bindings to existing GCS implementations like Appia, JGroups and Spread and other communication services like an IP-based multicast service and NeEM.

Unfortunately, this middleware architecture can only be statically configured and no dynamic reconfiguration or switching is possible for the moment.

6.6.2 Dynamic Switching Systems

Ensemble's ***Protocol Switch Protocol*** (van Renesse et al., 1998)

The Ensemble system [50] is a group communication system based on the configuration and use of a *stack of protocols*, as in its predecessor Horus [99]. Each protocol of the stack provides a different service (message transport, group membership, ordering, etc.) to the application or to other protocols of the stack.

In [97], the Protocol Switch Protocol (PSP) is proposed. The PSP is an Ensemble protocol that allows the dynamic replacement of the full protocol stack used by Ensemble.

The PSP is a two-phase commit protocol (2PC) [46, 62]. In the first phase, one of the participant nodes takes a coordinator role and broadcasts a `FINALIZE` message, to ask to all nodes to start a protocol stack replacement. This message includes the composition of the new protocol stack. Upon reception of the `FINALIZE` message, each node stops the protocols in its current protocol stack, builds up the new protocol stack and then sends back a `FINALIZE-ACK` message to the coordinator. When the coordinator has received all the acknowledgement messages, it starts the second phase.

In the second phase, the coordinator broadcasts a `START` message. When a node receives the `START` message, it discards its current protocol stack and starts the regular operation with the new protocol stack.

The protocol includes some fault-tolerance support that tolerates the loss of messages (by means of retransmissions) and the node failures or disconnections.

On the other hand, in the coarse description of the protocol in [97] no details are given about the guarantees needed to multicast the `FINALIZE` and `START` control messages. Moreover, nothing is said about the need to block incoming or outgoing messages.

The PSP presents a significant disadvantage. As it is composed of two independent parts and the second part is not started until the first one is completed, the regular operation of the application is somehow blocked. The fact that the whole protocol stack is replaced is actually another inconvenience. Indeed, there is no way to replace a single protocol in a given protocol stack without having to stop and replace all the protocols of the stack.

Protocol Switching Based on State Transformation (Liu et al., 2000)

In [63], the authors present a mechanism alternative to the switching protocols based on a 2PC technique. The idea is to make the switching more scalable, by avoiding the dependency on a single coordinator node and reduce the delay imposed by the transition from the older protocol to the new one. This alternative consists in defining *switching functions* that are used to switch from the state kept by a protocol to the state used by another protocol.

In run-time, during a dynamic protocol switching, the use of such functions allow the nodes to go on working with the new protocol, which starts by managing the messages *inherited* from the first protocol and then goes on with the new messages.

Ensemble's Second Switching Protocol (Liu et al., 2001)

In [64] a second *Switching Protocol* (SP) is presented. Unlike the protocol presented in [97], the SP allows the replacement of a single protocol of the Ensemble's protocol stack.

The protocol is presented as a *wrapping* protocol that sits on top of a number of alternative protocols that offer the same service, i.e. the same guarantees. This wrapping protocol offers those guarantees to the protocol layered about it, which, in fact, does not need to know about its *wrapping* nature. When it operates in *normal mode*, it just forwards up and down the messages sent by and delivered to its neighbor layers. When it operates in *switching mode*, it performs a protocol replacement. As in [97], the SP assumes some mechanism that decides about when the current protocol has to be changed. Thus, the protocol replacement starts when some *oracle* chooses a node as a replacement *manager*.

The protocol operation is similar to that of [97] but there are some differences. First of all, the communication among the manager and the rest of nodes is no longer based on broadcasts. Instead, a logical ring is formed among all nodes and a token is forwarded from node to node along the ring. The token has a *mode* field that identifies the phase of the protocol.

When no protocol change is being performed, the nodes forward a token whose mode is `NORMAL`. To start a protocol change, a manager node waits until it receives a `NORMAL` token. Then it changes the mode to `PREPARE` and forwards the token to the next node in the ring. When a node receives a `PREPARE` token, it saves in some field of the token the number of messages it has sent with the current protocol and then forwards the token. When the manager node receives back the `PREPARE` token, it contains the numbers from all nodes. Then, it changes the token to `SWITCH` and forwards it again.

When a node receives a `SWITCH` token, it gets the number of messages sent by each node. When the manager node receives the `SWITCH` token, it changes the token mode to `FLUSH` and then forwards it once more.

When a node receives the `FLUSH` token it waits until it has received all the messages sent by all nodes with the current protocol. Then, it changes the current protocol to the new one and forwards the token. When the `FLUSH` token is finally received by the manager node, the protocol replacement is finished.

This protocol has some drawbacks related to its *blocking* nature. First of all, it prevents nodes from sending messages with both the current and the new protocol until they are in the third token round. Indeed, as the new protocol is not started until the third round of the switching protocol, no messages can be sent using the new protocol until then[1]. Moreover, the structure of the protocol, based on three rounds along the ring imposes a significant delay. Furthermore, this delay is increased by the blocking third round.

To argue about the correctness of the protocol, in [64] the authors formulate six *meta-properties* (*safety*, *asynchrony*, *delayable*, *send-enabled*, *memory-less* and *composable*) which are properties that describe other properties. Then, they argue that the switching protocol *preserves* these *meta-properties*. In short, they argue that if two protocols (for instance, two total order protocols) offer some property P (for instance, a *Total Order* property), and P satisfies those six *meta-properties*, then the switching protocol *is* a protocol that in turn offers P. This is formally proved in [24], by means of the NuPRL theorem prover [8].

[1]Although it is not explicitly said in [97], it is assumed that once a protocol change is started, i. e. once a node receives a `PREPARE` token, the message sending with the current protocol is stopped.

Adaptive Architecture in Cactus (Chen et al., 2001)

In [32, 28], another adaptive architecture for run-time protocol switching is proposed. This architecture is designed for Cactus [22], a framework for building distributed protocols and applications.

As in other distributed middlewares and frameworks, a Cactus application is based on a stack of layered components and each one of these offers a service. Some of these components may be *adaptive*, which means that they include different implementations of the same service. Initially, one of the available implementations of a given component is chosen. This architecture allows to change, in run-time, the current implementation of a service to one of the other available implementations of the service, in order to adapt to changing environments or contexts. For this, each adaptive component also includes an *adaptor*, which is a module that collaborates with the service implementations to perform the replacement.

The protocol change procedure is actually an abstract generic protocol, composed of three phases. A first phase is the detection of some changing environment or application parameters. A second phase, closely related to the first one, includes the election of the new implementation of the service. As in the solutions proposed by other authors, very little detail about these phases is given.

The third phase is the *adaptation* phase, which in turns consists of three steps: a) preparation, b) *outgoing switchover* and c) *incoming switchover*. This is a general scheme and the basic idea is that any protocol change can be decomposed in these steps, regardless the kind and nature of the service implementations that are to be replaced.

The preparation step includes all the actions needed to start and prepare the switching from one implementation of the service to the new one. It finishes with a *synchronization barrier*. Once all the participating nodes reach this barrier, they can proceed with the next step. The outgoing switchover is the step by which the flow of outgoing messages that arrive to a service implementation are *redirected* to a different implementation of the service. The incoming switchover is a similar message *redirection*, applied to incoming messages.

The generic protocol change scheme is implemented in the adaptor module of the adaptive component. This module depends on the semantics and nature of the service implementations to replace. In [32], the replacement of the total order broadcast service is given as an example of an implementation of the general scheme. Basically, the replacement procedure is the one specified by the general scheme. A significant detail is that once reached the synchronization barrier, at the end of the first step and before performing the *outgoing switchover*, the outgoing messages that could not be sent with the previous total order protocol are broadcast, by means of the new total order protocol.

One of the main drawbacks of the solution presented in [32] is that it forces the service implementations to fulfill a given interface. Actually, this requirement is not too strong, since this adaptive architecture was designed for Cactus systems, that are already forced to follow this requirement. Anyway, such a drawback may be solved by means of additional indirection layers that could be placed on top of each particular service implementation, thus acting as *adaptors*.

In essence, our proposal shares the general idea behind this proposal and its *structure* organized in three different parts. The broadcast of the `PREPARE` message in the algorithm proposed in Section 6.2 can be compared to the *synchronization barrier* used in [32]. Moreover, both solutions need to queue the outgoing messages that could not be sent with the previous protocol and send them later, with the new protocol, once it has been activated.

Dynamic Protocol Update (Rütti et al., 2006)

In [88] the problem of *Dynamic Protocol Update* is considered, as a particular case of the more general *Dynamic Software Update* problem.

The solution proposed is based on two *switching algorithms* that allow the dynamic replacement of one of the protocols of the protocol stack used by a user application. There is a switching protocol to replace the consensus protocol of the stack and another switching protocol to replace the atomic broadcast protocol. This solution is aimed at the SAMOA framework [105] but the basic idea may be applied to other protocol stack-oriented frameworks. The goal of the architecture presented is to allow the dynamic replacement of software components, thus easing the software maintenance and upgrade tasks. Nevertheless, this architecture can also help to improve the performance of the applications, as proposed in the introduction of this chapter.

According to the architecture proposed, one of the switching protocols is placed in the protocol stack, just above the protocol to change. When no protocol change is to be done, the switching protocol simply forwards up and down the messages sent by and delivered to the application. During a protocol change, the switching protocol intercepts the application messages. The general idea of *interception* includes delaying and resending messages. Although some minor differences exist, the operation of the protocols is basically very similar. For instance, both algorithms guarantee that the *service requests* performed with the current protocol (consensus or atomic broadcast) are finished before starting the operation with the new protocol. A *service request* is either a consensus instance or the broadcast of a message, depending on the protocol to replace.

The operation of the atomic broadcast switching protocol actually relies on the atomic broadcast protocol to be replaced. When a node decides to start a protocol change, it broadcasts a special message with the current atomic broadcast protocol. When a node receives this special message, it performs the protocol replacement, by installing and activating the new protocol. If there are some pending messages sent with the old protocol they will be discarded by all nodes at delivery time and resent by their corresponding senders, using the new protocol. This way, the switching protocol avoids the need of an additional acknowledgment message round (as in other proposals like the presented in this section or the one presented in Section 6.2). As in other proposals, nothing is said about how is decided to start a protocol change or which criteria are considered.

In [88], a discussion of the properties guaranteed by the switching protocols is also provided. These properties are expressed in terms of *modules*, *services* and *module bindings*. A protocol stack is modelled as a stack of *modules*. Each module is configured as a provider of a *service* by means of a *module binding*. A binding can be done statically, in configuration time, or dynamically, during a protocol change.

First, two properties of the switching protocols are proposed. Both properties have a *strong* and a *weak* variant. The *stack well-formedness* property expresses the need to have all the services bound to any module. The strong variant of this property requires that if a service is invoked it must have been bound to any module. The weak variant of the property requires that if a service is invoked, it is *eventually* bound to any module.

The *protocol operationability* property requires the need to have the required module installed in a stack when a protocol change is issued. Informally, the strong variant of this property requires that if a module (protocol) is bound in the stack of a node, then the stacks of all nodes contain that module. The weak variant of the property requires that this binding is just eventually done.

In [88] (and also [106]), it is shown that the atomic broadcast switching protocol ensures the *strong stack well-formedness* and the *weak protocol operationability* properties. Moreover, it is also shown that the regular properties of the atomic broadcast protocols (*Validity*, *Uniform Agreement*, *Uniform Integrity* and *Uniform Total Order*) are preserved by the atomic broadcast switching protocol.

Finally, some performance evaluation of both protocols is also presented. This evaluation includes the analysis of the latency of a series of messages broadcast by a set of nodes, during which an atomic broadcast protocol replacement is requested. As shown in the graphical results, the need to resend some messages during the execution of the protocol change algorithm has a negative impact on the latency of a number of messages.

Mocito's Run-time Switching (Mocito et al., 2006)

In [79] another switching protocol for total order protocols is proposed. In essence it is very similar to the one presented in [69].

In particular, they share some relevant features. First, it avoids blocking message sending with the new protocol so the flow of application messages is never blocked. It also sets a point in time from which no more messages are sent with the current total order protocol. Moreover, incoming messages broadcast with the new protocol are queued until all the pending messages are delivered with the current total order protocol and the protocol switching is completed.

They differ in the way the participant nodes *learn* about when they must *deactivate* the current total order protocol. In [69], the nodes count the number of messages broadcast with the current protocol and when a protocol change is started, this information is spread so all nodes know how many messages have to be delivered with the current protocol before deactivating it. In [79], each node broadcast an acknowledgement message as the last message broadcast using the current total order protocol. Upon reception of all such acknowledgement messages, a given node knows that no more messages will be sent with the current total protocol so the node can deactivate it.

Broadcast Protocol Switching (Karmakar et al., 2007)

In [56], the authors deal with the use of a switching protocol to dynamically change the broadcast protocol used by a network of nodes. A broadcast protocol based on a Breadth-First Search tree yields lower message latencies when the network load is low. On the other hand, a broadcast protocol based on a Depth-First Search reduces the load on individual nodes when the global network load is higher.

The mechanism discussed in [56] can switch between two broadcast protocols, one based on a BFS tree and another based on a DFS tree. The core of the mechanism is the construction of the spanning tree used by the broadcast protocol. In the paper, a protocol is shown to build a new DFS spanning tree. Nevertheless, no protocol is shown to build a BFS spanning tree.

6.7 Conclusion

In this chapter we review the problem of dynamically replacing the total order broadcast protocol used by a distributed application. As a result, we provide a new, non-blocking, highly concurrent switching protocol, fully integrable with existing independent membership services. Moreover, this protocol admits concurrent starts of the switching procedure.

The chapter includes an extensive description of the switching protocol, a pseudocode algorithm and a discussion of the properties offered by the switching protocol that allow it to behave like a regular total order protocol. We also include an experimental evaluation of its operation.

Although this switching protocol was designed to allow the dynamic replacement of regular total order broadcast protocols, it can also be used to replace *prioritized* total order broadcast protocols, without any further modifications.

To argue about this, we must consider that the prioritized protocols we presented in Chapter 3, behave like regular total order protocols and that *Prioritization* is a property that can be observed on the sequence of messages they totally order. These protocols can be wrapped in an architecture like the one presented in Figure 6.1. As long as the order of the sequence of messages provided by a given GCP is preserved by this architecture, the *Prioritization* property will be preserved. Moreover, as the switching protocol only relies in the regular properties offered by common total order protocols (*Validity*, *Uniform Agreement*, *Uniform Integrity* and *Total Order*) and does not specifically rely on any other properties like *Prioritization*, it can be isolated from specific total order broadcast implementations and additional semantics offered by them.

Part III

Conclusion

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis provides a number of contributions on the *prioritized total order* topic.

As a first contribution, we present a number of *generic techniques* to add prioritization support to existing total order protocols. These techniques are *generic* in the sense that they are not concrete implementations in any programming language, based in some particular framework. Instead, they are described as *modifications* to the *classic* total order protocols. Thus, these techniques can be applied to other total order algorithms and protocols and even to particular implementations.

As a second contribution, we show the result of an experimental study we performed to assess the *effectiveness* of the proposed *prioritization techniques*, based on a *prototype implementation* of a number of *classic* total order protocols that we developed. This study allows us to assert that in some settings, the prioritization mechanisms we implemented can indeed offer a *benefit*. As we showed, this benefit is typically *application-dependent*. In our case study, the *benefit* is a reduction of the *transaction abort rate* in an application that *simulates* the usage of a replicated database where integrity constraints have been defined.

As a third contribution, we show a second experimental study, that allows us to assess the *overhead* imposed by the prioritization mechanisms, in terms of *message delivery delays*, *CPU overhead* and *memory consumption*. This study allows us to assert that the prioritization mechanisms do not impose a significant overhead to the original total order protocols to which they are applied.

The fourth contribution is the ***Switching protocol***, a protocol to *dynamically switch total order protocols.* This protocol allows an application to dynamically (i.e. in run-time) change the total protocol that it is being currently used by the application. The ***Switching protocol*** has a number of advantages over other total order switching protocols. First, it works with regular and prioritized total order protocols. Moreover, it is *non-blocking*, which means that, during its operation, it does not block neither the sending of application messages nor the delivery of messages to the application. Thus, during a *protocol switch*, the application can go on sending and receiving application messages in total order and even with *prioritization guarantees*, if prioritized total order protocols are used. In a third experimental study we show that the overhead imposed by the ***Switching protocol*** is minimal, even when a protocol switch is being carried on.

7.2 Open Issues and Future Work

This thesis provides a number of contributions on the *prioritized total order* topic. Nevertheless, there are some tasks that still remain open.

A first task may be to implement more conventional total order protocols and their corresponding prioritized versions. The experimental evaluation work may be extended by including the new protocols. It may be also extended by considering additional criteria. For instance, it may be interesting to check how the application sending patterns can influence the results.

Another task would be to apply the proposed techniques to existing implementations of total order protocols. The JGroups and Appia open source frameworks allow any user to modify the existing protocols and even develop new ones which may be included in the original framework. The prioritization techniques proposed in this thesis may be applied to the total order protocols included in these frameworks and the prioritized versions may be included in the original frameworks. The test applications may be rewritten in order to adapt them to the frameworks and then, the experimental evaluation may be repeated. This task would help us to validate and improve the testing methodology.

Moreover, regarding the ***Switching protocol***, it would also be interesting to *migrate* it to JGroups or Appia. First, it may be tested with just the original protocols in the selected framework. Then, it may be tested with the original protocols and their prioritized versions. These tests may contribute to the validation of the ***Switching protocol***.

In another dimension, the results got in this research encourage us to try to apply the ideas behind the ***Switching protocol*** to some other contexts. For instance, they could be applied to the design of a generic platform to allow the dynamic switch of conventional *binary code bundles.* A typical use of such a platform would be to reload a web application in an application server, for instance during development stages. This kind of *software hot-swapping* could even be integrated in a more general architecture-level platform, to allow the dynamic switching of any part of a regular software application.

Bibliography

[1] The Spread Toolkit: http://www.spread.org.

[2] JGroups website: http://www.jgroups.org.

[3] Appia: http://appia.di.fc.ul.pt.

[4] JBoss Netty, http://www.jboss.org/netty.

[5] JSR 51: New I/O APIs for the Java Platform, http://www.jcp.org/en/jsr/detail?id=51.

[6] Robert K. Abbott and Hector García-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, 1992.

[7] M. K. Aguilera and R. E. Strom. Efficient atomic broadcast using deterministic merge. In *19th Annual ACM International Symposium on Principles of Distributed Computing (PODC-19)*, pages 209–218, Portland, OR, USA, 2000.

[8] Stuart F. Allen, Rich Eaton, Christoph Kreitz, and Lori Lorigo. The nuprl open logical environment. In *17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes of Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[9] Ángel Álvarez, Sergio Arévalo, Vicent Cholvi, Antonio Fernández, and Ernesto Jiménez. On the interconnection of message passing systems. *Information Processing Letters*, 105(6):249–254, February 2008.

[10] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Dependable Systems and Networks*, pages 327–336, Washington, DC, USA, 2000. IEEE-CS.

[11] Yair Amir, Louise E. Moser, P. Michael Melliar-Smith, Deborah A. Agarwal, and P.Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, Nov 1995.

[12] Yair Amir and Ciprian Tutu. From total order to database replication. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria*, July 2002.

[13] Ozalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group membership and view synchrony in partitionable asynchronous distributed systems: specifications. Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, 40127 Bologna, Italy, 1996.

[14] Ozalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: specification and algorithms. Technical Report UBLCS-98-01, Department of Computer Science, University of Bologna, Bologna, Italy, April 1998.

[15] Ozalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: specification and algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, April 2001.

[16] T. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3(1):67–99, 1991.

[17] Roberto Baldoni, Roberto Beraldi, Roy Friedman, and Robbert van Renesse. The hierarchical daisy architecture for causal delivery. *Distributed Systems Engineering*, 6(2):71–81, 1999.

[18] Roberto Baldoni, Stefano Cimmino, Carlo Marchetti, and Alessandro Termini. Performance analysis of Java group toolkits: a case study. In *FIDJI '01: Revised Papers from the International Workshop on Scientific Engineering for Distributed Java Applications*, pages 49–60, London, UK, 2003. Springer-Verlag.

[19] Ziv Bar-Joseph, Idit Keidar, Tal Anker, and Nancy Lynch. Qos preserving totally ordered multicast. In *5th International Conference on Principles of Distributed Systems (OPODIS 2000)*, pages 143–162, 2000.

[20] Ziv Bar-Joseph, Idit Keidar, and Nancy A. Lynch. Early-delivery dynamic atomic broadcast. In *16th International Conference on Distributed Computing (DISC'02)*, pages 1–16, London, UK, 2002. Springer-Verlag.

[21] Piotr Berman and Anupam A. Bharali. Quick atomic broadcast. In *7th International Workshop on Distributed Algorithms (WDAG'93)*, pages 189–203, London, UK, 1993. Springer-Verlag.

[22] Nina T. Bhatti. *A system for constructing configurable high-level protocols.* PhD thesis, Department of Computer Science, The University of Arizona, Dec. 1996.

[23] Nina T. Bhatti and Richard D. Schlichting. A system for constructing configurable high-level protocols. In *SIGCOMM*, pages 138–150, 1995.

[24] Marck Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving hybrid protocols correct. *Lecture Notes in Computer Science*, 2152/2001:105–120, 2001.

[25] Ken Birman and Robert van Renesse, editors. *Reliable distributed computing with the Isis toolkit.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[26] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, Austin, Texas, United States, 1987. ACM Press.

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

[28] Patrick G. Bridges, Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Supporting coordinated adaptation in networked systems. In *Eigth Workshop on Hot Topics in Operating Systems*, 2001.

[29] Nuno Carvalho, José Pereira, and Luís Rodrigues. Towards a generic group communication service. In *Distributed Objects and Applications International Conference (DOA'06)*, volume 4276 of *Lecture Notes in Computer Science*, pages 1485–1502, 2006.

[30] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distibuted systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[31] M. Chen and K. Lin. Dynamic priority ceiling: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2(1):325–346, 1990.

[32] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 635–643, Mesa, Arizona, USA, 2001.

[33] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[34] Gregory V. Chockler, Nabil Huleihel, and Danny Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symposium on Principles of Distributed Computing*, pages 237–246, Puerto Vallarta, Mexico, 1998. ACM Press.

[35] The GORDA Consortium. Group communication service in sourceforge.net. http://jgcs.sourceforge.net.

[36] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118:158–179, 1995.

[37] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Real-Time Systems*, 2(3):195–212, 1990.

[38] Xavier Défago, André Schiper, and Péter Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, 2003.

[39] Xavier Défago, André Schiper, and Péter Urbán. Totally ordered broadcast and multicast algorithms: taxonomy and survey. Technical Report IS-RR-2003-009, École Polytechnique Fédérale de Lausanne, Sept. 2003.

[40] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[41] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[42] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[43] Alan Fekete, Nancy Lynch, and Alex Shvarstman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, 2001.

[44] Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[45] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):48–59, January 1982.

[46] Jim Gray. Notes on database operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[47] Rachid Guerraoui and Andre Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[48] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed systems*, chapter 5, pages 97–145. ACM Press, Addison-Wesley, 2nd edition, 1993.

[49] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Department of Computer Science, University of Toronto; Department of Computer Science, Cornell University, 1994.

[50] Mark Hayden. *The Ensemble System.* PhD thesis, Cornell University, 1998.

[51] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[52] N. F. Jo-Mei Chang and Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[53] Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of mutual inconsitency in distributed systems. In *Berkeley Workshop*, pages 172–184, 1981.

[54] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the amoeba distributed operating system. In *11th International Conference on Distributed Computing Systems (ICDCS'91)*, pages 222–230, April 1991.

[55] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the Amoeba group communication system. In *16th IEEE International Conference on Distributed Computing Systems (ICDCS '96)*, pages 436–448, Washington, DC, USA, 1996. IEEE Computer Society.

[56] Sushanta Karmakar and Arobinda Gupta. Adaptive broadcast by distributed protocol switching. In *ACM symposium on Applied computing (SAC'07)*, pages 588–589, New York, NY, USA, 2007. ACM.

[57] Idit Keidar and Danny Dolev. *Dependable Network computing*, chapter Totally ordered broadcast in the face of network partitions, pages 51–75. Kluwer Academic, 1999.

[58] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 156–163, May 1998.

[59] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1018–1032, July/Aug. 2003.

[60] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, , and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, 1997.

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

[62] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, June 1979.

[63] Xiaoming Liu and Robbert van Renesse. Fast protocol transition in a distributed environment (brief announcement). In *19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, page 341, New York, NY, USA, 2000. ACM.

[64] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In Luís Rodrigues and Michel Raynal, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*. IEEE CS Press, 2001.

[65] Shyw-Wei Luan and Virgil D. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):271–285, 1990.

[66] Pattie Maes. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12):147–155, 1987.

[67] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.

[68] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan 48824, July 2004.

[69] Emili Miedes, Mari-Carmen Bañuls, and Pablo Galdámez. Group Communication Protocol Replacement for High Availability and Adaptiveness. In *Advanced Distributed Systems: 6th International School and Symposium (ISSADS)*, Guadalajara, México, January 2006.

[70] Emili Miedes and Francesc D. Muñoz-Escoí. Adding priorities to total order broadcast protocols. Technical Report TR-ITI-ITE-07/23, Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, October 2007.

[71] Emili Miedes and Francesc D. Muñoz-Escoí. Reducing transaction abort rates with prioritized atomic multicast protocols. Technical Report TR-ITI-ITE-07/22, Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, October 2007.

[72] Emili Miedes and Francesc D. Muñoz-Escoí. Managing Priorities in Atomic Multicast Protocols. In *International Conference on Availability, Reliability and Security (ARES)*, pages 514–519, Barcelona, Spain, March 2008. ISBN 0-7695-3102-4.

[73] Emili Miedes and Francesc D. Muñoz-Escoí. On the cost of prioritized atomic multicast protocols. Technical Report ITI-SIDI-2009/002, Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Feb. 2009.

[74] Emili Miedes and Francesc D. Muñoz-Escoí. On the cost of prioritized atomic multicast protocols. In *11th International Symposium on Distributed Objects, Middleware and Applications (DOA 2009)*, pages 585–599, Vilamoura, Portugal, November 2009. Lecture Notes in Computer Science (LNCS), vol. 5870, pages 585–599, Springer-Verlag, Heilderberg (Germany). ISBN 978-3-642-05147-0.

[75] Emili Miedes and Francesc D. Muñoz-Escoí. Dynamic total-order broadcast protocol replacement. Technical Report ITI-SIDI-2010/001, Instituto Universitario Mixto Tecnológico de Informática, Universidad Politécnica de Valencia, March 2010.

[76] Emili Miedes, Francesc D. Muñoz-Escoí, and Hendrik Decker. Reducing Transaction Abort Rates with Prioritized Atomic Multicast Protocols. In *14th International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 394–403, Las Palmas de Gran Canaria, Spain, August 2008. Lecture Notes in Computer Science (LNCS), vol. 5168, pages 394–403, Springer-Verlag, Heilderberg (Germany). ISBN 978-3-540-85450-0.

[77] Emili Miedes and Francesc D. Muñoz-Escoí. Dynamic switching of total-order broadcast protocols. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 457–463, Las Vegas, Nevada, USA, July 2010. CSREA Press. ISBN 1-60132-158-9.

[78] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia: a flexible protocol kernel supporting multiple coordinated channels. In *21st International Conference on Distributed Computing Systems*, pages 707–710, 2001.

[79] José Mocito and Luís Rodrigues. Run-time switching between total order algorithms. In *EuroPar 2006*, 2006.

[80] Tim Moors. A critical review of 'end-to-end arguments in system design'. In *IEEE International Conference on Communications*, pages 1214–1219, 2002.

[81] Louise E. Moser, Yair Amir, P. Michael Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

[82] Louise E. Moser, P. Michael Melliar-Smith, Deborah A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[83] Akihito Nakamura and Makoto Takizawa. Priority-based total and semi-total ordering broadcast protocols. In *12th International Conference on Distributed Computing Systems (ICDCS 92)*, pages 178–185, June 1992.

[84] Akihito Nakamura and Makoto Takizawa. Starvation-prevented priority based total ordering broadcast protocol on high-speed single channel network. In *2nd International Symposium on High Performance Distributed Computing*, pages 281–288, July 1993.

[85] Fernando Pedone. *The database state machine and group communication issues.* PhD thesis, École Polytechnique Fédérale de Lausanne, 1999.

[86] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, Jan. 1994.

[87] Luís Rodrigues, Paulo Veríssimo, and Antonio Casimiro. Priority-based totally ordered multicast. In *3rd IFAC/IFIP workshop on Algorithms and Architectures for Real-Time Control (AARTC'95)*, Ostend, Belgium, May 1995. IFAC.

[88] Olivier Rütti, Pawel Wojciechowski, and André Schiper. Structural and algorithmic issues of dynamic protocol update. In *20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[89] Seyed Masoud Sadjadi. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan 48824, 2003.

[90] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[91] André Schiper, Kenneth Birman, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[92] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6):43–48, June 2002.

[93] Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, 1993.

[94] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[95] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[96] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, 1992.

[97] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. *Software Practice and Experience*, 28(9):963–979, 1998.

[98] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in Horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, 1995.

[99] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: a flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[100] Yun Wang, Emmanuelle Anceaume, Francisco Brasileiro, Fabíola Greve, and Michel Hurfin. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, 51(8):900–915, August 2002.

[101] Yun Wang, Francisco Brasileiro, Emmanuelle Anceaume, Fabíola Greve, and Michel Hurfin. Avoiding priority inversion on the processing of requests by active replicated servers. In *Dependable Systems and Networks*, pages 97–106. IEEE Computer Society, 2001.

[102] Matthias Wiesmann. *Group Communications and Database Replication: Techniques, Issues and Performance*. PhD thesis, Faculté I&C, Section D'Informatique, École Polytechnique Fédérale De Lausanne, 2002.

[103] Matthias Wiesmann, Xavier Défago, and André Schiper. Group communication based on standard interfaces. In IEEE, editor, *2nd IEEE International Symposium on Network Computing and Applications (NCA-03)*, 2003.

[104] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Database replication techniques: a three parameter classification. In *19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, pages 206–215, 2000.

[105] Pawel T. Wojciechowski, Olivier Rütti, and André Schiper. SAMOA: a framework for a synchronisation-augmented microprotocol approach. *18th IEEE Parallel and Distributed Processing Symposium (IPDPS2004)*, April 2004.

[106] Pawel T. Wojciechowski and Olivier Rütti. On correctness of dynamic protocol update. In Springer LNCS 3535, editor, *7th IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS05)*, June 2005.