UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA


# Run-Time Switching Between Total Order Algorithms

**José Carlos Vitório Mocito**

MESTRADO EM INFORMÁTICA

2006

# Run-Time Switching Between Total Order Algorithms

José Carlos Vitório Mocito

Dissertação submetida para obtenção do grau de
MESTRE EM INFORMÁTICA

pela

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

**Orientador:**

Luís Eduardo Teixeira Rodrigues

**Júri:**

Rodrigo Miragaia Rodrigues

José Manuel de Sousa de Matos Rufino

Luís Manuel Pinto da Rocha Carriço

2006

# Resumo

Os protocolos de ordem total são elementos fundamentais na construção de muitas aplicações distribuídas e tolerantes a faltas. Infelizmente, a sua implementação pode ser dispendiosa, quer no número de passos de comunicação, quer na quantidade de mensagens trocadas. Este problema é ainda mais evidente em redes de grande-escala, onde o desempenho do algoritmo pode ser limitado pela presença de ligações com latência elevada. Para reduzir este problema foram propostos diversos protocolos de ordem total optimistas. No entanto, os serviços prestados por cada um destes protocolos divergem entre si, e cada protocolo oferece desempenhos distintos consoante as propriedades da rede onde se executa.

Esta dissertação apresenta uma descrição das várias aproximações optimistas e estabelece uma caracterização das suas propriedades e adequação a vários ambientes de execução.

Um protocolo adaptativo que permite a comutação dinâmica entre diferentes algoritmos de ordem total é proposto e avaliado. O protocolo possibilita a execução do algoritmo mais favorável a cada momento, permitindo obter o melhor desempenho possível. Os resultados experimentais mostram que a solução proposta permite a troca de algoritmos com uma interferência desprezável na comunicação.

**PALAVRAS-CHAVE:** Ordem Total, Ordem Total Optimista, Algoritmo Adaptativo.

# Abstract

A total order protocol is a fundamental building block in the construction of many distributed fault-tolerant applications. Unfortunately, the implementation of such a primitive can be expensive both in terms of communication steps and of number of messages exchanged. This problem is exacerbated in large-scale systems, where the performance of the algorithm may be limited by the presence of high-latency links. Optimistic total order protocols have been proposed to alleviate this problem. However, different optimistic protocols offer quite distinct services. Moreover, there are certain algorithms that perform better in specific scenarios and given network properties.

This dissertation provides an overview of different optimistic approaches and establishes a characterization of their properties and suitability to different execution environments.

An adaptive protocol that is able to dynamically switch between different total order algorithms is proposed and evaluated. The protocol allows to achieve the best possible performance by supporting the reconfiguration such that, in each moment, the algorithm that is most appropriate to the present network conditions can be executed. Experimental results show that, using our protocol, adaptation can be achieved with negligible interference in the data flow.

**KEY WORDS:** Total Order, Optimistic Total Order, Adaptive Algorithm.

# Acknowledgments

My first acknowledgment goes to my professor and supervisor, Professor Luís Rodrigues, for his dedication to all the aspects that surrounded the execution of this work. His prosecution for excellence has been a continuous inspiration and challenge.

To my parents and grandparents my biggest acknowledgment, for the infinite support I always received. May this work reflect all the confidence you showed in my capabilities, and fill your hearts with great joy.

To my other half, Mónica, I thank the objectiveness she provides to my life and the permanent incentive I received every single day in the course of this past months. Thank you for always believing in me.

I thank all my family, in which I obviously include the Ribeiro family, for the friendship and confidence they always demonstrated.

To all my friends, with no exception, I am grateful for all the great relaxing moments we shared.

I thank all my colleagues at LaSIGE, and particularly the members of the DIALNP research group, with whom I shared lots of interesting discussions that contributed significantly to the outcome of this dissertation.

Finally, this work was partially supported by the IST project GORDA (FP6-IST2-004758). I would also like to thank the LaSIGE research unit and the Department of Informatics at FCUL, for the working conditions they

provided for the prossecution of this dissertation.

Lisboa, June 2006

José Carlos Vitório Mocito

*Ao meu Avô Francisco.*

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

A total order broadcast protocol is a fundamental building block in the construction of many distributed fault-tolerant applications (Powell, 1996). Informally, the purpose of such a protocol is to provide a communication primitive that allows processes to agree on the set of messages they deliver and, also, on their delivery order. Uniform total order broadcast is particularly useful to implement fault-tolerant services by using software-based replication (Guerraoui & Schiper, 1997).

Unfortunately, the implementation of such a primitive can be expensive both in terms of communication steps and number of messages exchanged. This problem is exacerbated in large-scale systems, where the performance of the algorithm may be limited by the presence of high-latency links. Several total order protocols have been proposed that use different strategies to offer good performance (Défago *et al.*, 2004). There is, however, no protocol that outperforms all others in all scenarios: each protocol offers best results under different load profiles and/or network conditions.

To allow for faster execution of services that rely on total order algo-

rithms an early estimation of the final order can be determined. This estimation, which we will refer to as *optimistic delivery*, can be used conditionally, to allow progress of the computation in parallel with the communication steps. However, an inaccurate estimation may lead to expensive rollback operations, thus voiding the advantage of delivering early.

The estimation accuracy of such optimistic algorithms depends greatly on the characteristics of the network where they execute. Even if the algorithm is chosen appropriately, these conditions may not hold for some periods of time. This motivates for the importance of executing the most suitable algorithm at any given time. Several solutions have been proposed to allow a component to change its behavior based on the execution environment. Some of these methods were applied to total order algorithms, however as we will show, they exhibit some limitations that may render them useless in critical applications, where significant disruptions in the commucation flow may be unacceptable.

## 1.1 Objectives

The first objective of this dissertation is to establish the usefulness of considering different types of optimistic deliveries and provide results that substantiate the impact of the different assumptions these protocols make in the timeline of these deliveries.

The second objective is to describe and evaluate a total order protocol that combines different algorithms and adapts itself to the running environment. The protocol is modular, preserving the total order semantics of the underlying algorithms and allowing its execution on top of existing total order broadcast implementations, without modifications.

## 1.2 Results and Main Contributions

During the course of accomplishing the abovementioned objectives the following results where obtained:

- An implementation of the adaptive total order protocol;
- The integration of the protocol in a simulation environment to allow faster experimentation and evaluation tests;
- A detailed performance evaluation of the adaptive total order protocol.

The main contributions of the dissertation are the following:

1. Identifies and classifies previously published works on total order as different degrees of "optimism" for the same final and authoritative total order broadcast service;
2. Proposes a novel adaptive total order protocol that combines different algorithms and adapts itself to the running environment, in a non-disruptive fashion to the ongoing communication.

## 1.3 Research History

This MSc dissertation is the result of a year's work as a researcher in the IST project GORDA (FP6-IST2-004758). The primary goal of the project is to develop an open database replication architecture. The database state-machine approach was chosen to give such support. In this approach, a group communication total order primitive is used to broadcast messages between replicas. The work in this dissertation fulfills a significant part of

the optimized group communication support required by the project. This optimized infrastructure will make use of the most suitable optimistic total order protocol to offer improved performance. For that we need a switching protocol that is able to switch between optimistic implementations, which is one of the main contributions of this dissertation.

Parts of the work described have been previously published in peer-reviewed international conferences. More specifically, parts of Chapter 3 have been published in the proceedings of the 21st Annual ACM Symposium on Applied Computing (Rodrigues *et al.*, 2006). Also, parts of Chapters 4, 5 and 6 have been accepted for publication in the proceedings of the Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference (Mocito & Rodrigues, 2006).

Meanwhile, some further progress as been made after the conclusion of the work discussed here, and we have already submitted for review a paper that provides the formal mechanisms to optimize sequencer-based optimistic total order protocols and proposes a practical algorithm that outperforms competing approaches.

## 1.4 Dissertation Structure

This dissertation is structured as follows.

Chapter 2, "Total Order Broadcast", overviews the related work regarding total order broadcast protocols. It introduces the specifications of the different variations of these protocols, with particular focus on the optimistic variations, presents the most common implementations and surveys the state-of-the-art in adaptive total order protocols.

Chapter 3, "Ranking Total Order Deliveries", discusses the usefulness

of combining different optimistic deliveries to provide better overall system performance. Several evaluation results are provided to support the discussed claims.

Chapter 4, "Run-time Switching Protocol", describes an adaptive total order protocol with a complete specification and an informal explanation of all the algorithm steps.

Chapter 5, "Implementation", provides a complete description of an implementation of the switching protocol.

Chapter 6, "Evaluation and Optimization", presents and discusses the results from the performance evaluation tests performed and briefly describes an implementation level optimization.

Chapter 7, "Conclusion", concludes the dissertation with an overall analysis of the major contributions and provides some insights on planned future research efforts.

# Chapter 2

# Total Order Broadcast

Informally, total order broadcast is a group communication primitive that ensures that messages sent to a set of processes are delivered by all those processes in the same order. Such a primitive is useful, for example, in the implementation of fault-tolerant services (Powell, 1996), for instance, using the state machine approach (active replication) (Schneider, 1990). This chapter focuses on three different aspects of total order broadcast. In Section 2.1 we present the properties of total order broadcast algorithms. Three widely used algorithms are described in Section 2.2, and Section 2.3 presents three different optimistic approaches for total order. Finally, in Section 2.4, we provide a survey on adaptive protocols that can be applied to total order algorithms.

| |
|---|
| **RTO1 - Regular Total order**: Let $m_1$ and $m_2$ be two messages that are *RTO-broadcast*. Let $p_i$ and $p_j$ be any two correct processes that *RTO-deliver*$(m_1)$ and *RTO-deliver*$(m_2)$. If $p_i$ *RTO-delivers*$(m_1)$ before *RTO-delivers*$(m_2)$, then $p_j$ *RTO-delivers*$(m_1)$ before *RTO-delivers*$(m_2)$, and we note $m_1 < m_2$. |
| **RTO2 - Agreement**: If a correct process in $\Omega$ has *RTO-delivered*$(m)$, then every correct process in $\Omega$ eventually *RTO-delivers*$(m)$. |
| **RTO3 - Termination**: If a correct process *RTO-broadcasts*$(m)$, then every correct process in $\Omega$ eventually *RTO-delivers*$(m)$. |
| **RTO4 - Integrity**: For any message $m$, every correct process *RTO-delivers*$(m)$ at most once, and only if $m$ was previously *RTO-broadcast* by some process $p \in \Omega$. |

Table 2.1: Regular total order properties

## 2.1 Specifications

### 2.1.1 Regular Total Order

Regular total order broadcast is defined on a set of processes $\Omega$ by the primitives (1) *RTO-broadcast*$(m)$ which issues message $m$ to $\Omega$, and (2) *RTO-deliver(m)* which is the corresponding delivery of $m$. When a process $p_i$ executes *RTO-broadcast*$(m)$ (resp *RTO-deliver(m)*), we say $p_i$ *"RTO-broadcasts m"* (resp *"RTO-delivers m"*). The total order primitive characterized by the properties listed in Table 2.1 is known as regular total order. Informally, a regular total order protocol ensures that two correct processes (i.e., processes that never crash) deliver exactly the same set of messages in the same order.

### 2.1.2 Uniform Total Order

A stronger version, called uniform total order (Défago *et al.*, 2004), can also be defined. It can be obtained by replacing properties *RTO1* and *RTO2* by

**UTO1 - Uniform Total order**: Let $m_1$ and $m_2$ be two messages that are *UTO-broadcast*. Let $p_i$ and $p_j$ be any two processes that *UTO-deliver*($m_1$) and *UTO-deliver*($m_2$). If $p_i$ *UTO-delivers*($m_1$) before *UTO-delivers*($m_2$), then $p_j$ *UTO-delivers*($m_1$) before *UTO-delivers*($m_2$), and we note $m_1 < m_2$.

**UTO2 - Uniform Agreement**: If a process in $\Omega$ (correct or not) has *UTO-delivered*($m$), then every correct process in $\Omega$ eventually *UTO-delivers*($m$).

Table 2.2: Uniform total order properties

properties *UTO1* and *UTO2* presented in Table 2.2. Uniform total order is stronger as it ensures that, if a processes $p_i$ delivers two messages in a given order, all processes will deliver the same messages in that order, *even* if $p_i$ fails. Uniform total order is the desired consistency criteria in applications such as database replication, given that certain messages may cause a transaction to be aborted or committed. If the delivery of a message to a process causes this process to commit a transaction before crashing, all other processes need also to deliver the same message to ensure a consistent outcome of the transaction.

### 2.1.3 Optimistic Total Order

The level of coordination required by regular and uniform total order protocols often makes them utterly expensive. Still, this service is required by many distributed systems. To circumvent performance limitations some authors have proposed optimistic approaches. An optimistic protocol provides early estimations of the final total order with the purpose of allowing applications to execute some steps in parallel with the communication steps of the total order protocol. These processing steps can later be committed or aborted when the final definitive order is established. Nat-

urally, the earlier the optimistic delivery can be provided, the more steps can be executed in parallel. However, the speedup gains obtained from this parallelism, can be compromised by the need to abort steps, when the estimate proves to be inaccurate.

An *optimistic* total order protocol includes an additional primitive *TO-opt-deliver(m)*. When a process $p_i$ executes *TO-opt-deliver(m)*, we say that $p_i$ *"TO-opt-delivers m"*. The order by which a process $p$ *TO-opt-delivers* messages is an estimate of the order by which $p$ will *TO-deliver* the same messages. Note that in some cases the estimate may be wrong, *i.e.*, the order by which messages are *TO-opt-delivered* may differ from the order by which they are *TO-delivered* (although in stable periods it is desirable that it is the same). Note also that it is possible that a message is directly *TO-delivered* without ever being *TO-opt-delivered*.

## 2.2 Total Order Algorithms

Many algorithms exist to implement total order. To give the reader an insight on the possible alternatives we briefly introduce three of the most used ones, namely the *sequencer* (Kaashoek & Tanenbaum, 1991), the *token-site* (Chang & Maxemchuck, 1984) and the *symmetric* (Peterson *et al.*, 1989; Dolev *et al.*, 1993) approaches.

Several other alternatives exist. For a comprehensive survey, the reader is referred to (Défago *et al.*, 2004). However, from the three examples below, it should be clear that it is interesting to have a protocol that can dynamically adapt to changes in the operation envelope by switching, in run-time, from one algorithm to another.

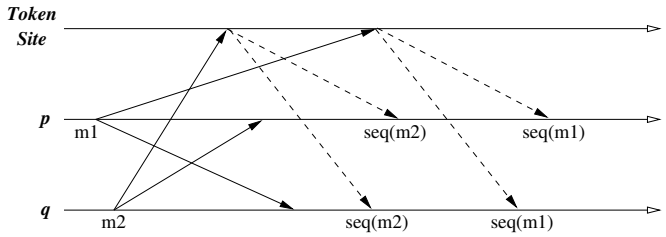


Figure 2.1: Sequencer-based total order algorithm.

### 2.2.1 Sequencer-based Total Order

In the sequencer-based approach (see Figure 2.1) one process is responsible for ordering messages on behalf of the other processes in the system. This process works as a sequencer of all messages and is often called the *sequencer* process. Sequencer-based algorithms are appealing because they are relatively simple and provide good performance when message transit delays are small (they are particularly well suited for local area networks). However, in a sequencer-based algorithm, a message sent by a process other than the sequencer experiences a delivery latency close to $2D$, where $D$ is the message transit delay between two system processes (i.e., the time to disseminate the message plus the time to obtain an order number from the sequencer). Thus, sequencer-based approaches are inefficient in face of large network delays.

### 2.2.2 Token-site Total Order

It is possible to design solutions where the sequencer role is rotated among processes. The main motivation for such solutions is to provide load-balancing among processes. These algorithms are often called token-site approaches, because at each moment in time only one process in the communication group holds the token. This process acts as a sequencer until it

transmits the token to another process. The delivery latency of such algorithms is very close to the sequencer-based approach however, depending on how quickly the token rotates, it may equal the time to disseminate the message plus the time to obtain the token from another process.

### 2.2.3 Symmetric-based Total Order

In the symmetric approach, ordering is established by all processes in a decentralized way, using information about message stability. This approach usually relies on *logical clocks* (Lamport, 1978) or *vector clocks* (Birman & Joseph, 1987b; Peterson *et al.*, 1989): messages are delivered according to their partial order and concurrent messages are totally ordered using some deterministic algorithm. Symmetric protocols have the potential for providing low latency in message delivery when all processes are producing messages. In fact, symmetric protocols can exhibit a latency close to $D+t$, where $t$ is the largest inter-message transmission time (Rodrigues *et al.*, 1996). Unfortunately, this also means that all (or at least a majority (Dolev *et al.*, 1993)) processes must send messages at a high rate to achieve low protocol latency.

### 2.2.4 Uniform Versions

All the algorithms described provide a regular total order service. However, obtaining the uniform version of the algorithms is not difficult. To preserve uniformity each process must be certain that a majority of the total number of processes in the gorup has received the message and is capable of delivering it to the application. For this to happen every process in the majority must already possess the regular order for that given

message.

In practice, ensuring uniformity translates in an added communication step, where processes share with each other the set of messages they have regularly ordered. When enough information is shared to reach a majority for a given set of messages, those messages can be delivered uniformly.

## 2.3 Optimistic Total Order Algorithms

### 2.3.1 Spontaneous Total Order

The notion of optimistic total order was first proposed in the context of local-area broadcast networks (Pedone & Schiper, 1998). In many of such networks, the spontaneous order of message reception is the same in all processes. Moreover, in sequencer-based total order protocols the total order is usually determined by the spontaneous order of message reception in the sequencer process. Based on these two observations a process may estimate the final total order of messages based on its local receiving order and, therefore, provide an optimistic delivery as soon as a message is received from the network.

### 2.3.2 Statistically Estimated Total Order

Spontaneous total order is, however, improbable in wide area networks. The long latency in wide-area links causes different processes to receive the same message at different points in time. Consider the topology depicted in Figure 2.2. Assume that process $a$ multicasts a message $m_1$ and that, at the same time, the sequencer $s$ multicasts a message $m_2$. Clearly, the sequencer will receive $m_2 < m_1$, given that $m_1$ would require $12ms$ to

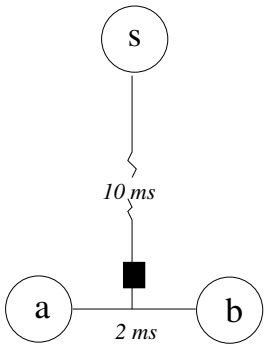


Figure 2.2: A network with local and wide-area links.

reach the sequencer. On the other hand, process $b$ will receive $m_1 < m_2$, as $m_1$ will take only $2ms$ to reach $b$ while $m_2$ will require $12ms$. From this example, it should be obvious that the spontaneous total order provided by the network at $b$ is not a good estimate of the observed order at the sequencer.

To address the problem above, (Sousa *et al.*, 2002) proposed to introduce artificial delays in the message reception to compensate for the differences in the network delays. It is easier to describe the intuition of the protocol by using a concrete example. Consider again the network of Figure 2.2. Assume also that we are able to provide to each process an estimate of the network topology and of the delays associated with each link. In this case, $b$ could infer that message $m_1$ would take $10ms$ more to reach $s$ than to reach $b$. By adding a delay of $10ms$ to all messages received from $a$, it would mimic the receive order of $a$'s messages at $s$. A similar reasoning could be applied to messages from other processes.

Informally, the algorithm consists in predicting the order by which the sequencer process receives messages from the network by exploiting local clocks and the stability in network transmission delays. The basic mechanism behind the solution is to observe both the spontaneous order and

the order received from the sequencer, and adjust the optimistic delivery of future messages accordingly. Each process only needs the information about the time between two consecutive messages and the time between their respective sequencer messages, to determine which messages need to be delayed in order to provide an order similar to the one observed in the sequencer process.

### 2.3.3 Optimistic Regular Total Order

Another type of optimistic delivery can be accomplished by any regular total order algorithm, as the ones presented in Section 2.2, as long as the desired final order is uniform. In this scenarios, regular total order always happens before the uniform one, because the later needs one extra communication step. The regular order is however optimistic, because it is possible that some failed process has delivered the message, without it being delivered by the remaining correct processes (and thus violating uniform agreement).

## 2.4 Adaptive Total Order Protocols

In distributed systems where a total order service must provide good performance, choosing the most suitable protocol is crucial. However, the choice of the most appropriate protocol may vary with certain conditions, like network delays, congestion or processing capacity. It is thus possible that, in a given environment, different total order algorithms provide better performance at different moments in time.

Adaptive total order protocols provide the ability to execute the more suitable algorithm at any given moment, by providing the mechanisms to

switch from one algorithm to the other in a straightforward matter and ideally, independent from the application layer.

To better understand such protocols, one needs to realize the challenges involved in their construction:

- The adaptation phase should be transparent to the application. This allows for proper modularization, making this protocols useful to virtually all applications that use common total order services;
- The switching phase should provide the less disruption possible, allowing for faster transitions with minimal impact in ongoing communication;
- Adaptation should provide consistent states in all elements of the communication group at all time. The same is to say that all group members must be executing the same or compatible total order algorithms;
- The previous goal is only feasible with some degree of coordination, which raises scalability issues. This coordination also raises efficiency problems, so it must be kept to a minimum.

Several adaptive total order protocols have been proposed to tackle the above challenges. Each protocol follows a different approach which differentiates it from the others. In order to establish a comprehensive comparison between them, a careful organization of their characteristics is presented in this section.

The next four sections describe each protocol in detail, highlighting the most important facts about each solution. Each section starts with an

overview of the protocol followed by an informal explanation of its behavior. In the end, a short discussion of the algorithm is provided along with a list of comparable characteristics. The last section summarizes all the information provided in the former sections by providing an organized view of each protocols characteristics.

Each protocol will be analyzed using five different vectors: implementation, scope of the adaptation, reconfiguration unit, level of coordination and adaptation smoothness. The implementation vector refers to the software execution environment where the protocol has been implemented. The scope of the adaptation relates to the potential of each solution to work with different kinds of services/protocols, i.e. if it suits only single services like total order protocols, or if it suites a broader range of services, like for instance distributed agreement protocols. The reconfiguration unit defines the subject of intervention when the adaptation takes place, i.e. the organizational unit that is the focus of adaptation. The level of coordination deals with the functional aspect of the adaptation procedure and defines the potential of scalability of each solution. The more decentralized/distributed the solution, the more scalable it is. Finally, the adaptation smoothness defines the level of communication disruption produced by the switching protocol. The possible values for this metric are explained in Table 2.3.

At this point the reader should be advised that each of these vectors is analyzed in the specific scenario of adaptive total order. Several of the following protocols are generic adaptive protocols that can be used to adapt services other than total order. When used with those services the characteristics described may not prevail. For instance, the switching protocol may include optional steps that may be skipped in certain conditions,

| Value | Description |
|---|---|
| Disruptive | Explicit traffic stoppage; often implies a message stabilization step |
| Semi-smooth | Implicit traffic stoppage or delay; often implies buffering |
| Smooth | Perfectly smooth in certain conditions; no buffering, only traffic delays |
| Perfect | Perfectly smooth in all conditions |

Table 2.3: Adaptation smoothness values.

which may produce totally different results.

### 2.4.1 Renesse *et al.*, 1998

#### Overview

The first solution analyzed was proposed by Renesse *et al.* in (van Renesse *et al.*, 1998). Their work describes a generic methodology to build adaptive systems on top of Ensemble (Hayden, 1998).

Ensemble is a protocol composition and execution framework that has been tailored to support group communication. The framework provides the facilities necessary for the development highly specialized protocols (called *micro-protocols*) that can be grouped in communication stacks. These protocols interact with each other using events, that flow up or down in the stack.

Although this system promotes the development of highly modular protocols, like for instance a total order protocol built on top of a virtually-synchronous group communication service, the adaptation approach does not allow for the isolated substitution of these entities. Instead, the whole stack must be replaced by a new one, that contains the new, more suitable protocols.

**Protocol**

The protocol switching protocol is based on a simple two-phase-commit approach. The protocol assumes that both stacks involved in the adaptation procedure may not be compatible in respect to the message headers they process. For this reason each stack is uniquely identified by a *Protocol Stack Instance Identifier* (PSI-ID). Also, this assumption enforces the stabilization of all the pending activities in the old stack, which also requires some form of coordination. This coordination is performed by the node with the lowest network address.

In general terms, the protocol can be defined by this tree steps:

1. Distribute and instantiate the new stack in each participant

2. Finalize the micro-protocols in the old stack

3. Start using the new stack

First, the coordinator broadcasts a FINALIZE message containing the description of the new stack, the group membership and the new PSI-ID. This message is received by all group members, including the coordinator. Upon reception each node instantiates the new stack and registers the new PSI-ID to allow new messages to be handled by the new stack. The FINALIZE event is put on top of the stack and is sent down, traversing all the layers. Each layer must hold the FINALIZE event until it stops sending messages. When this happens it passes the event to the layer below. When it reaches the end of the stack a FINALIZE-ACK is sent to the coordinator, signaling the finalization of the old stack. When all the FINALIZE-ACK messages are received from all members of the group the coordinator broadcasts a START message to the new stack. When each

node receives such message it discards the old stack and restarts the normal execution using the new stack. Figure 2.3 illustrates these interactions.

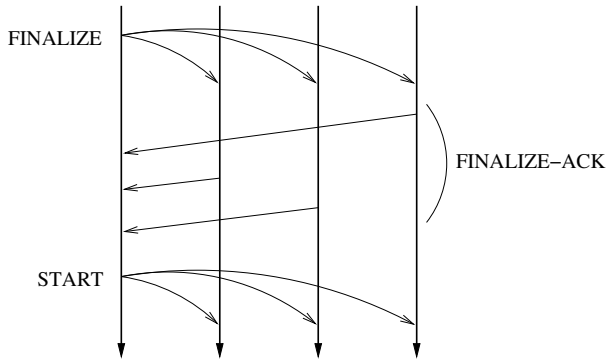


Figure 2.3: Renesse et al. switch protocol

**Discussion**

This protocol has the obvious advantage of relying in a very clean technique that provides a generic enough mechanism to support most kinds of adaptations. However, by relying on a single coordinator this approach is prone to scalability issues. The efficiency of the protocol is also limited by the slowest node in the communication group. Another important negative fact is the finalization procedure that promotes a drastic disruption in the communication flow. All traffic must be stopped in the old protocol before the new one can be used, which does not allow a very smooth adaptation.

**Characteristics**

| | |
|---|---|
| **Implementation:** | Ensemble |
| **Scope:** | Generic adaptive protocol |
| **Reconfiguration unit:** | Whole communication stack |
| **Level of coordination:** | Centralized coordinator |
| **Adaptation smoothness:** | Disruptive |

### 2.4.2 Liu & Renesse , 2000

**Overview**

In a brief announcement (Liu & van Renesse, 2000) Liu & Renesse describe an algorithm that provides a fast protocol transition in a distributed environment. The negative aspects of two-phase-commit algorithms as the one described in the previous section motivated this proposal, which performs protocol switching with little overhead and works in a decentralized fashion.

The solution relies on the assumption that for two protocols derived from the same abstract specification *AS*, there exists two converting functions that transform the state of one protocol to the other. This makes possible for the existence on one hybrid protocol that makes adaptations between algorithms that derive from the same *AS*.

**Protocol**

To allow for smooth adaptation this protocol uses a buffering mechanism and a fast transition procedure. The algorithm work in three steps:

1. One process initiates the switching procedure by broadcasting a special “switch” message

2. When a "switch" message is received a process stops the current protocol and starts buffering application messages. It also sends out all the information required for all processes to convert their local states

3. When all the needed information is gathered, each process converts its local state to that of the new protocol and starts using it

The proposal also describes the application of this algorithm to the switching between two types of total order broadcast: sequencer-based $S$ and token-based $T$ protocols. In this context the algorithm works as follows. The state of $S$ consists of a buffer holding the messages still to be ordered by the sequencer. In $T$ the state is a buffer containing the messages to be sent when in possession of the token. To switch from $S$ to $T$ the sequencer broadcasts the identifiers of messages already ordered from all processes. All the processes then copy the messages in the buffer on $S$ to the buffer on $T$. In the inverse operation, from $T$ to $S$, the process that holds the token sends the ordered information to the sequencer, while all the other processes copy messages from the buffer on $T$ to the one on $S$ and send them to be sequenced.

**Discussion**

The adaptive algorithm has the main advantage of being completely decentralized. Also, the state conversion procedure can be performed very fast when compared to communication stabilization procedures. On the downside, the adaptation procedure is tightly coupled with the conversion functions, which must exist for every supported protocol. Most importantly, it still requires some stoppage in the message flow, due to the buffering methodology.

**Characteristics**

| | |
|---|---|
| **Implementation:** | Not available |
| **Scope:** | Generic adaptive protocol |
| **Reconfiguration unit:** | Protocol |
| **Level of coordination:** | Totally distributed |
| **Adaptation smoothness:** | Semi-smooth |

### 2.4.3 Chen *et al.*, 2001

**Overview**

Chen *et al.* have also proposed a protocol (Chen *et al.*, 2001) with the clear goal of providing a *graceful adaptation* mechanism, where the underlying adaptive process is completely hidden and transparent to the application. The proposal is also generic, as it applies to every distributed algorithm and not specifically to total order.

Like the Renesse et al. method described in Section 2.4.1, the protocol assumes a distributed system model where the software contained in every node consists of multiple modular components grouped in layers. However, it clearly distinguishes itself by allowing the adaptation to happen at the layer level, instead at the whole stack. This allows for finer-granularity in the adaptation process and does not halt message exchange during the switch between algorithms.

The algorithm was implemented using Cactus (Hiltunen & Schlichting, 2000) which, like Ensemble, allows the development of highly configurable services through the composition of modular entities that interact with each other using events. A service is defined by a set of micro-protocols that are grouped in a stack and define a set of event handlers that they use to interact with other micro-protocols or with the Cactus

run-time system.

One difference to Ensemble is that Cactus provides binding (and unbinding) mechanisms that allow a protocol to register (or unregister) its event handlers at execution time, thus allowing the dynamic activation (or deactivation) of these entities.

**Protocol**

The protocol that provides the adaptation procedure is, once more, quite simple and involves three steps:

1. Preparation of the new protocol
2. Application level starts using the new protocol to send messages
3. Incoming messages start being processed by the new protocol

Each step executes several actions before terminating. The preparation step consists of instantiating the new protocol and execute a global synchronization barrier to guarantee that all nodes have completed the instantiation. In the second step, if required or useful, some state may be transferred between the old and new protocols. The new protocol is then registered for handling outgoing messages from the application, instead of the old one. At this point, if necessary, the new protocol starts buffering outgoing messages, to preserve the ordering properties of the messages exchanged. The step finishes when all pending actions related to the old protocol are completed. Finally, in the last step the new protocol processes all its buffered outgoing messages (if they exist) and the system resumes its normal execution.

The application of this algorithm to total order is straightforward. The first and last steps are executed as described. The intermediate step must

have the buffering action properly activated because we are dealing with a protocols that must preserve message ordering. Before terminating it must ensure that all messages sent using the old protocol are properly ordered. Only then the last step is executed.

#### Discussion

This protocol has the clear advantage of allowing the adaptation to occur in a single layer instead of in all the stack. It also provides several degrees of flexibility during the switchover, like the *optional* state transfer or buffering of outgoing messages, that in some cases can be omitted and allow for faster termination. Nevertheless, for the specific case of total order protocols, the algorithm still induces a stoppage in the message flow, as the buffering needs to take place to preserve the ordering properties of the service.

#### Characteristics

| | |
|---|---|
| **Implementation** | Cactus |
| **Scope** | Generic adaptive protocol |
| **Reconfiguration unit** | Protocol |
| **Level of coordination** | Global synchronization barrier |
| **Adaptation smoothness** | Semi-smooth |

### 2.4.4 Rutti *et al.*, 2006

#### Overview

All the above approaches are generic enough to support all kinds of transitions between distributed protocols. However, their application to adaptive total order protocols always results in stoppages in the communica-

tion flow, due to the necessity of first terminating the old algorithm before using the new one. In the paper (Rutti *et al.*, 2006) Rutti *et al.* proposed a *dynamic protocol update* algorithm for switching between distributed agreement protocols. This work defines a common set of properties that must remain unchanged during every reconfiguration process.

The system model proposed defines an architecture that allows for transparent application execution despite the adaptive characteristics of the underlying system. Each protocol implements a service, and several services can be combined in communication stacks. Each service can be implemented by several protocols, which may or may not be bound to the service. The binding of protocols to services can be done dynamically, however unbinding a protocol does not remove it from the stack. It just "disconnects" it from the service, but remains in the stack for future "connections". The protocols interact with each other by calling services, which implicitly corresponds to calling the above or bellow protocols that implement those services. As you can see this model has some similarities with the ones used in the algorithms of Sections 2.4.1 and 2.4.3, although with some nuances, like for instance allowing a straightforward binding and unbinding of protocols in a stack.

#### Protocol

The approach followed in this proposal is to define replacement modules that contain the adaptation logic for specific services. The protocols that implement these services are required to maintain valid a static set of properties, which allows these replacement modules to handle every protocol that implements such service. These modules also serve as an intermediary between the underlying protocols and the above applications or ser-

vices, impersonating the role of the underlying protocols to allow transparency.

In the paper the dynamic protocol replacement was illustrated by two algorithms, one for replacing consensus protocols and the other for replacing atomic broadcast protocols. Atomic broadcast is a communication primitive equivalent to uniform total order. In the context of this survey we will thus describe the former replacement algorithm.

In the algorithm every node keeps track of messages it sent but have not been ordered. To initiate the adaptation process a control message is broadcast to all processes to initiate the reconfiguration. When a node receives this message it substitutes the old protocol with the new one with the unbinding and binding operations described. It then checks for messages it sent but have not yet been ordered and re-broadcasts them using the new protocol. After this point all the messages received in the old protocol are simply discarded and the normal execution is resumed.

**Discussion**

As you can see this algorithm has the clear advantage of not requiring any centralized coordination, and most importantly of allowing an immediate transition to the new protocol without any buffering required. The negative aspect of this algorithm is that the average latency during the switching phase increases. The magnitude of this increase depends on the send-rate of the node and its consequent accumulated messages that are still to be delivered in order. In certain load conditions this effect may clearly translate into a perceptible stoppage in the communication flow.

**Characteristics**

| | |
|---|---|
| **Implementation:** | SAMOA |
| **Scope:** | Distributed agreement protocols |
| **Reconfiguration unit:** | Protocol |
| **Level of coordination:** | Totally distributed |
| **Adaptation smoothness:** | Smooth |

## 2.5 Summary

The previous sections provided a detailed explanation of each protocol and highlighted several important characteristics of each one. However, such information is spread across the description of the protocols in a non-organized way. In Table 2.4 we provide a summary of this information in a straightforward and comprehensive way. In each column is represented a protocol and is analyzed following the evaluation vectors considered throughout this chapter: implementation, adaptation scope, unit of reconfiguration, level of coordination and adaptation smoothness.

| | **Renesse** | **Liu** | **Chen** | **Rutti** |
|---|---|---|---|---|
| **Implementation** | Ensemble | N/A | Cactus | SAMOA |
| **Scope** | Generic | Generic | Generic | Agreement Prot. |
| **Reconf. unit** | Stack | Protocol | Protocol | Protocol |
| **Coordination** | Centralized | Distributed | Sync. Barrier | Distributed |
| **Smoothness** | Disruptive | Semi-smooth | Semi-smooth | Smooth[1] |

Table 2.4: Characteristics of adaptive total order protocols.

[1] Depends heavily on the message load. With sufficient heavy load can be significantly disruptive.

# Chapter 3

# Ranking Total Order Deliveries

To illustrate the trade-offs involved in an optimistic total order protocol, we will use the most intuitive algorithm to establish total order: the sequencer based algorithm. Note that a similar discussion could be made using other total order algorithms, but the simplicity of the sequencer approach makes the text more clear. In a sequencer based algorithm, one of the processes in the system, designated the sequencer, has the onus of assigning a sequence number to every message it receives. All processes, including the sequencer, deliver messages according to these sequence numbers.

To better describe the steps involved in an uniform total order proto-

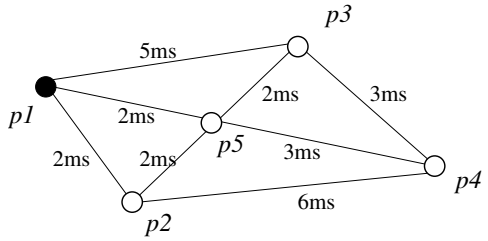


Figure 3.1: Network with 5 nodes.

col, we will use the network illustrated in Figure 3.1. The figure shows a network with five nodes $p_1 - p_5$ connected by point-to-point links. The average delay of each link is also depicted (for instance, the average delay in the link $p_1 - p_3$ is $5ms$).

Let us now consider a particular run using the network above. This run is depicted in Figure 3.2. At time $t_0 = 0$ process $p_2$ sends a message $m_2$ and process $p_3$ sends a message $m_3$. Assume that process $p_4$ receives message $m_3$ at time $t_3 = 3$ and message $m_2$ at time $t_6 = 6$. We name the order by which messages are received at each process from the underlying transport protocols the *spontaneous order* (SO).

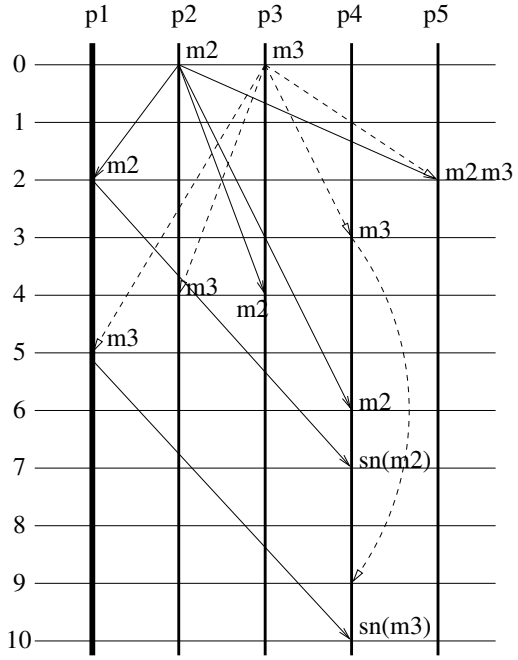


Figure 3.2: A run.

In the same example, assume that $m_2$ is received by process $p_1$, the sequencer, at time $t_2 = 2$ and $m_3$ at time $t_5 = 5$. Assume that the sequencer assigns sequence number to messages in the order it receives them. Clearly,

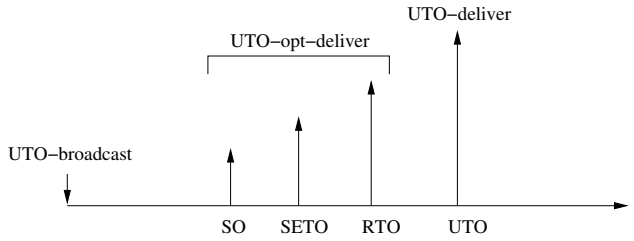


Figure 3.3: Optimism rank.

in this example, the spontaneous order observed at $p_4$ would be different than the final order as assigned by the sequencer. Sequence numbers assigned by the sequencer will be received at process $p_4$ at times $t_7 = 7$ and $t_{10} = 10$. In this case, as soon as the sequence number is received we have assured *regular total order* (RTO). Note that if both $p_1$ and $p_4$ fail, it is still possible that the remaining processes assign a difference order to messages $m_2$ and $m_3$.

The final *uniform total order* (UTO) can only be guaranteed when $p_4$ is sure that the sequencer numbers have been received by all the remaining processes (or, at least, a majority). In our example this would happen at time $t_{13} = 13$.

Note that, if $p_4$ can estimate that the delay between $p_1$ and $p_3$ is $3ms$ higher than the delay between $p_1$ and $p_2$, it could attempt to reproduce the order by which the sequencer receives messages $m_2$ and $m_3$ by artificially delaying the delivery of $m_3$ by a delta of $6ms$ (i.e., by delivering $m_3$ at time $t_9 = 9$). A clever scheme inspired in this insight has been proposed to establish a *statistically estimated total order* (SETO) before the regular total order is known (Sousa *et al.*, 2002).

Figure 3.3 shows a timeline of total order delivery. Naturally, spontaneous order occurs first in the timeline and uniform total order occurs last.

The question now is to decide which of the intermediate orders should be used to support optimistic delivery.

Clearly, spontaneous total order is only an accurate indication of the final delivery if all nodes use the same local area network segment, as assumed in (Pedone & Schiper, 1998). On the other hand, the statistically estimated total order is a good choice in stable networks, where the network delays have small variance and can be accurately estimated. In unstable networks, regular total order is the best choice for optimistic delivery, as it only provides inaccurate information in the rare cases where a node crashes.

## 3.1 Experimental Results

We have performed a number of experiments to validate the comparative behavior of *Statistically Estimated Total Order* (SETO) and *Regular Total Order* (RTO) protocols. The experiments were made using the SSFNet network simulator (Nicol *et al.*, 2003). The network topology used consists on a wide area network with two clouds, connected by one link. Each cloud contains one router that is used to connect the clouds. The other nodes form a group membership. Figure 3.4 shows the network used in the experiments.

The average latency in the long-haul connecting the two networks is 20*ms*; this is the major source of latency in this experimental setting. In order to simulate instability in the network the standard deviation of the transmission delays of the long-haul link is made variable between 0% to 10%.

Every node in the simulation receives messages from the sending group

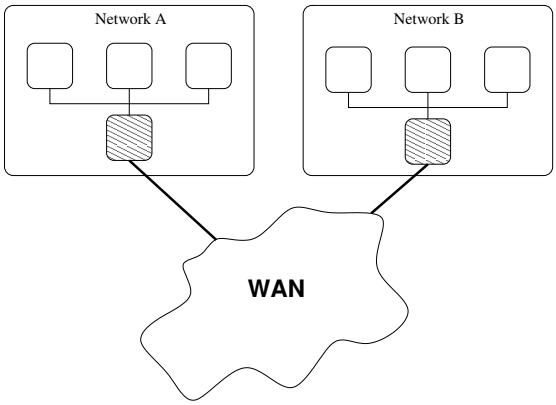


Figure 3.4: Network used in the simulation.

members. Senders transmit messages at a variable, uniformly distributed rate. In Network A we have two nodes that actively send messages while in Network B there is only one sender. The sequencer is located in Network A. All values depicted in the figures and tables below were measured at a node located in Network B.

This configuration was chosen to illustrate the behavior of SETO protocol in particular conditions. As said before, spontaneous order is only accurate when all nodes execute on the same local area network segment. By having sender nodes on both networks the spontaneous delivery will become an inaccurate estimation of the final delivery. This scenario thus make the case for other types of optimistic deliveries. The second aspect that we want to illustrate is the impact of the variation in the transmission delays in the performance of the SETO protocol. By having two senders in Network A we will be able to observe that some messages sent by these nodes will exchange their delivery order when traversing the long-haul link in their way to Network B producing, once more, an inaccurate estimation of the final delivery at B.

As described in Section 2.1.3, optimistic delivery is only useful if highly

| Type of Delivery | Error Rate |
|---|---|
| Spontaneous | 66% |
| SETO | 13% |

Table 3.1: Error Rate in Stable Network

accurate, i.e., if the application is not required to rollback the execution steps performed optimistically very often. Table 3.1 clearly shows that in heterogeneous networks (i.e., in networks were nodes have different distances to the sequencer) the spontaneous order in an unacceptable source for optimistic delivery, as the error rate is extremely high even in a stable network where the standard deviation is 0%. On the other hand, the SETO approach, in a stable network, can provide a significantly smaller error rate and provides an interesting source of optimistic delivery.

In Figure 3.5, we show the impact of network instability on the accuracy of SETO. As can be seen, an increase in the standard deviation of the transmission delay in the long haul link makes the error rate of the SETO protocol increase, reaching 40% for a $\sigma = 10\%$. These results confirm the results published in (Sousa *et al.*, 2002) and clearly show that SETO accuracy is highly dependent on the network stability.

Table 3.2 shows the timeline of deliveries in our experiment. The values are averages of all messages received by our target node in Network B. Naturally, spontaneous order provides the smaller latency, given that messages are delivered as soon as they are received from the network. The value depicted in the table can be explained as follows. We recall that the measurements are made in a received in Network B. There are two senders in Network A and their messages suffer and average delay of $20ms$. The messages from sender located in Network B suffer a negligible delay. Since all senders transmit at approximately the same average rate, the average

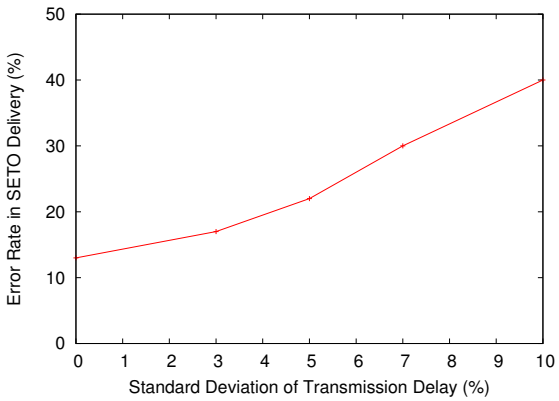


Figure 3.5: Error Rate in SETO Delivery

| **Type of Delivery** | **Time of Delivery** |
|---|---|
| Spontaneous | 13429 $\mu s$ |
| SETO | 20891 $\mu s$ |
| Regular | 40835 $\mu s$ |
| Uniform | 42528 $\mu s$ |

Table 3.2: Type of TO Delivery *vs* Time of Delivery

delay becomes approximately 13$ms$. A similar reasoning explains the figures depicted in the remaining rows. The interesting aspect is that SETO offers significantly less latency than regular delivery. Therefore, in stable networks SETO is the source of choice for providing optimistic delivery. Also, regular delivery is still faster than the final uniform delivery and provides room for optimistic execution of application steps.

## 3.2 Summary

In this chapter we showed how the SETO protocol provides a fast and accurate estimation of the final total order in stable networks. Unfortunately, in unstable networks, SETO delivery is highly inaccurate, produc-

ing a very high rate of rollbacks. In such networks a RTO delivery is better suited, because of its robustness regarding variability in the transmission delays.

# Chapter 4

# Run-time Switching Protocol

In this chapter we describe a run-time switching protocol that allows a smooth transition between total order algorithms. We provide the motivational context that resulted in the development of the protocol and present in detail the algorithm.

## 4.1 Overview

By now it should be clear that every algorithm proposed for switching between total order protocols may result in communication disruption during the reconfiguration phase. The motivation behind our proposal is to develop an algorithm that does not suffer from this problem, or at least avoids it in specific conditions.

We now propose a protocol that is able to switch from a total order algorithm to another total order algorithm in response to changes in the operation envelope (such as changes in the workload, network conditions, number of participants, etc). In this dissertation we do not focus on the conditions that trigger adaptation, as these are highly application depen-

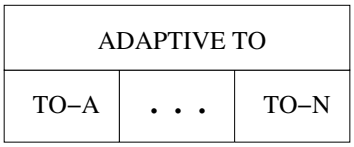


Figure 4.1: Adaptive protocol.

dent (for a concrete scenario, see (Rodrigues *et al.*, 2006)). Instead, we are interested in finding a generic switching procedure that can switch from one algorithm to the other with minimum interference in the data flow.

Such protocol can be built from scratch using a monolithic approach where all the functionality of every total order algorithm is embedded in a single unity. A more modular (and generic) way of reaching the same goal is to (re-)utilize independent implementations of total order algorithms and build the adaptive behavior on top of them as depicted in Figure 4.1. In steady-state, the adaptive protocol would simply receive *TO-broadcast*/*TO-deliver* requests/indications and forward them to the most appropriate algorithms.

As described in Section 2.4, previous works on dynamic adaptation requires messages to be buffered during the reconfiguration (Liu & van Renesse, 2000), the message flow to be stopped in the current protocol (van Renesse *et al.*, 1998), or some communication delay to be imposed during the transition between protocols (Rutti *et al.*, 2006). Here we describe a generic transition protocol that does not require the traffic to be stopped, allowing a smooth adaptation to changes in the underlying network.

To be able to effectively transition from one algorithm to the other, all nodes need to agree on the point in the message flow where they switch. The rational behind our proposal is to start broadcasting messages using both total order algorithms, even before the switching point is reached in

```
 1: Initialization:
 2:     deliv ← ∅
 3:     undeliv ← ∅
 4:     curAlg ← TO-A {current algorithm}
 5:     newAlg ← ∅ {next alg.}
 6:     switching ← false
 7:     check[1..n] ← false
 8: upon changeAlgorithm(newTO) do
 9:     rBroadcast(switch,newTO)
10: upon rDeliver(switch,newTO) do
11:     newAlg ← newTO
12:     switching ← true
13:     TO-broadcast(curAlg,(flag,null,myself))
14: upon TO-deliver(curAlg,(flag,null,sender)) do
15:     check[sender] ← true
16: upon check[1..n] = true do
17:     endSwitch()
```

Figure 4.2: Adaptive Total Order algorithm (Part 1)

every process. By using both algorithms simultaneously, no stoppage in the message flow is necessary.

## 4.2 Algorithm

The protocol listed in Figures 4.2 and 4.3 works as follows. Let us assume that the adaptation protocol is using algorithm TO-A to order messages and wants to switch to algorithm TO-B. The transition protocol works as follows. A control message is broadcast to all processes to initiate the reconfiguration (lines 8–9). When a node receives this message (line 10) it starts broadcasting messages using both total order algorithms. Also, the first message it broadcasts using algorithm TO-A is flagged. If no message is to be sent, then a flagged special null message is broadcast using

```
18: upon TO-broadcast(msg) do
19:     TO-broadcast(curAlg,msg)
20:     if switching = true then
21:         TO-broadcast(newAlg,msg)

22: upon TO-deliver(alg,msg) do
23:     if alg = curAlg ∧ msg ∉ deliv then
24:         deliver(msg)
25:         deliv ← deliv ∪ {msg}
26:     else if msg ∉ deliv then
27:         undeliv ← undeliv ∪ {msg}

28: procedure endSwitch()
29:     for all msg ∈ undeliv ∧ msg ∉ deliv do
30:         deliver(msg)
31:         deliv ← deliv ∪ {msg}
32:     undeliv ← ∅
33:     check[i..n] ← false
34:     curAlg ← newAlg
35:     switching ← false
```

Figure 4.3: Adaptive Total Order algorithm (Part 2)

TO-A, to allow faster protocol termination (flagged first message is not represented in the algorithm to preserve clarity). When a process starts receiving messages from both TO algorithms it performs the following steps (lines 22–27): messages received from TO-A are delivered as normally; messages received from TO-B are buffered in order. As soon as a flagged message is received from each and every node (line 15) the transition is concluded using the following "sanity" procedure (lines 28–35). Firstly, all messages received from TO-B that have not yet been delivered by TO-A are delivered in order. Finally, from this point on, all messages received from TO-A are simply discarded and no further message is sent using TO-A (until a new reconfiguration is needed). The TO-B algorithm is then used to broadcast and receive all the messages to be delivered.

Note that, after the transition is concluded, messages received from

TO-B are delivered only if they have not been already received and delivered from TO-A (line 23). This is a necessary safeguard as the two total order algorithms do not necessarily deliver messages in the same order, nor at the same time. So there is a possibility that a message that has already been delivered from TO-A is received after the termination of the reconfiguration procedure from TO-B.

Also, the protocol presented does not allow concurrent adaptations. For one adaptation to happen, the previous (if any) should always have concluded.

## 4.3 On Failure Detection

To simplify the description of our protocol, in previous sections we have not addressed the issue of failure detection. Namely, we have stated that the protocol moves to the sanity step when it receives a flag from *every* participant (Figure 4.2, line 16). Without further changes, the protocol would simply block in the presence of a single failure. We now discuss how our protocol can be adapted to operate under different failure models. Our algorithm can operate in asynchronous systems augmented with failure detectors (Chandra & Toueg, 1996).

We start by discussing the operation of the protocol in a system augmented with a *Perfect Failure Detector* ($\mathcal{P}$) (Chandra & Toueg, 1996), i.e., a system where processes fail by crashing and crashes can be accurately detected by all correct processes. In this model, the transition condition should be set to "a flag is received by all *correct* processes". This model is actually used in all of our implementations, where the failure detection is encapsulated by a view-synchronous interface (Birman & Joseph, 1987a).

The protocol can also be modified to operate in an asynchronous system augmented with an unreliable failure detector (such as the $\diamond S$ failure detector proposed in (Chandra & Toueg, 1996)) as long as a majority of processes do not fail (naturally, in this case, the underlying total order algorithms must also be designed for such a model). In this model, the transition condition should be set to "a flag is received by a majority of processes". However, in this configuration, correct processes that do not belong to the majority may be required to retransmit some messages.

It is interesting to observe that the strategy proposed before for the $\mathcal{P}$ detector (perform the switch when a flag is received from *all* correct processes) and the strategy proposed in (Rutti *et al.*, 2006) (perform the switch when the *first* flag is received) can be seen as extreme points of a spectrum (see Figure 4.4). Between these extreme cases, there is a range of alternative switching points, from which the "majority of processes" is the one that ensures less disruption in $\diamond S$ model.

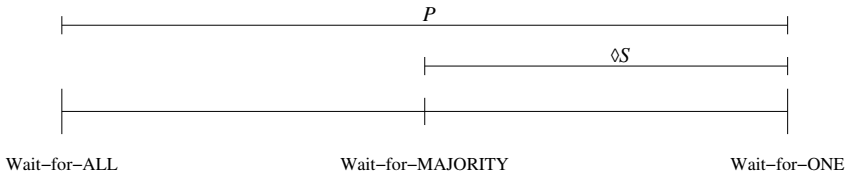


Figure 4.4: Wait-condition choice spectrum.

## 4.4 Summary

In this chapter we presented a run-time switching algorithm that provides support for switching between different total order algorithms with reduced impact in the communication flow. A specification of the algorithm

is provided along with an informal explanation of its execution. Finally, we discussed the failure detection assumptions of the algorithm and provided some modifications that allow its execution with an unreliable failure detector.

# Chapter 5

# Implementation

In order to validate the switching protocol we implemented it in Java using the *Appia* middleware framework. This chapter provides an introduction to this framework followed by a description of the switching protocol, which includes both organizational diagrams and excerpts from the source code of the most significant parts of the implementation.

## 5.1 Appia

*Appia* (Miranda *et al.*, 2001) is a framework that supports the implementation and execution of modular protocol compositions (see Figure 5.1). Each *Appia* module is a layer, i.e., a micro-protocol responsible for providing a particular communication service. These layers are independent and can be combined. A combination of layers constitutes a protocol stack that offers a given quality of service, QoS for short (in the broad sense of QoS, encompassing reliability, security, etc).

Once a QoS has been defined, by composing the appropriate layers, it is possible to create one or more communication channels. To each chan-

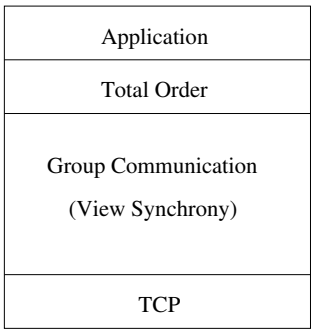


Figure 5.1: *Appia* stack example.

nel is associated a stack of *sessions*: for each protocol layer there is a session responsible for maintaining the state required for the execution of the corresponding protocol. Two channels that share a given layer may share the same session (see Figure 5.2). In this case, the protocol may correlate events exchanged in different channels with the help of the state maintained by the shared session. For instance, if two different channels share a session of a causal order protocol, messages exchanged by these channels are ordered among each other.

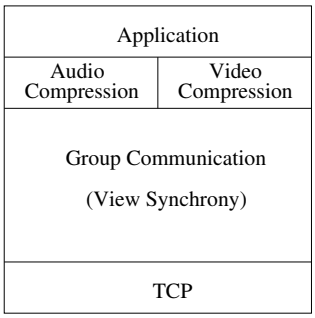


Figure 5.2: *Appia* shared session example.

Layers interact through the exchange of events. Events are typed and each layer is responsible for declaring which types of events it wants to process and which type of events the layer creates. Using this information the *Appia* system automatically optimizes the flow of events in the stack.

The *Appia* distribution provides an *out of the box* reliable group communication protocol suite. Available group communication services include membership services, reliable multicast services, view-synchrony, ordering services (causal and total order), among others.

## 5.2 Design Issues

The implementation of the switching protocol in *Appia* took into account the following design issues:

1. Be compatible with all the total order protocols already bundled with *Appia*;

2. Be transparent to the above applications and/or protocols;

3. Comply with the specification of the algorithm as described in Section 4.2.

## 5.3 Failure Detection Assumptions

As discussed in Section 4.3, the proposed run-time switching protocol assumes the presence of a perfect failure detector. The implementation of the protocol that we describe in this chapter makes use of a view-synchronous group communication service to abstract this assumption. Each failure in the system is reported to the protocol as a view change, where the new view contains only the correct processes in the group.

## 5.4 Implementation Assumptions

Because the focus of this research is the switching procedure, some assumptions were made during the implementation in order to simplify the problem and to allow a faster assessment of the evaluation results. Firstly, the implementation is limited to the switching between two fixed total order algorithms. Secondly, we assume that these algorithms are provided in advance, and not provisioned during the adaptation phase.

None of the above assumptions is relevant to the validation or evaluation of the switching procedure. They are both tightly related to the coordination of the adaptation process, in particular with the components that provide the decisions for when and what to reconfigure, which are not the subject of analysis in this dissertation.

## 5.5 Stack organization

The communication stack we used to validate and evaluate the switching protocol is depicted in Figure 5.3. The top-level application is simply illustrative and may, or may not be part of the stack. However, its presence in the stack simplifies development and testing.

The stack consists of two different channels, with some sessions shared between them, namely the application, coordination, group communication, NakFIFO and UDP sessions. Each channel holds one different total order session. The group communication suite is a set of micro-protocols that implements the functionality of a view-synchronous group communication service. NakFIFO is a protocol that implements FIFO order using negative acknowledgments. The bottom layer is an implementation of the UDP protocol. The layer that remains unexplained is the switching proto-

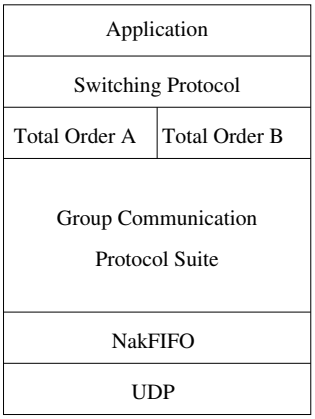


Figure 5.3: Communication stack organization.

col which implements the run-time switching protocol that is described in Chapter 4.

All the sessions other than the switching protocol may, or may not be shared between the channels. They are shared for resource optimization reasons, but it is not mandatory. However, the switching session must be shared among both channels so it can coordinate the adaptation process and redirect the application events to the proper total order protocol.

Despite the fact that the focus of research was not on the coordination of the reconfiguration, a simple protocol that signaled the switching protocol to initiate a switchover had to be implemented in order to experiment with the switching protocol. Such protocol was positioned between the application and the switching protocol, as depicted in Figure 5.4. Its implementation is rather irrelevant to the main focus of discussion and so we will not delve any further into this subject.

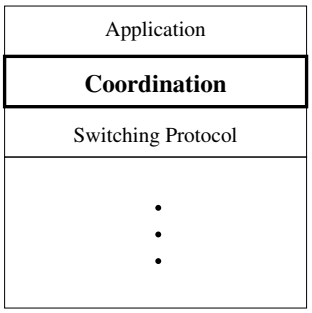


Figure 5.4: Coordination layer.

## 5.6 Switching protocol

In this section we will give a detailed explanation of the implementation of the switching protocol. We will use small source code snippets extracted from the actual implementation, usually comprising a single method, to guide the reader through the whole description.

Some of the code presented is a stripped down version of the actual implementation. The purpose of this deliberate omission is to allow a better understanding of the core features of the implementation, and not distract the reader with implementation details that are not related in any way with the actual protocol specification.

### 5.6.1 Initialization

We start by presenting the first part of the algorithm, the switchover initialization. As described in the algorithm presented in Section 4.2, one process is responsible for broadcasting a control message that signals the beginning of the switching procedure. When this message is received the initialization method `startReconfiguration()` is called (see Listing 5.1). In lines 2–7 all the properties related to the switchover are properly reset to their initial values. Next, a timer associated with the null message

```
1  private void startReconfiguration() {
2    switching = true;
3    checked = 0;
4    check = new boolean[vs.view.length];
5    isFirstMsg = true;
6    nullFirst = false;
7    msgFirst = false;
8
9    NullEventTimer nullt = new NullEventTimer(NULL_TIMEOUT, "NullEventTimer",
10     currentChannel, Direction.DOWN, this, EventQualifier.ON);
11   nullt.go();
12 }
```

Listing 5.1: Switchover initialization.

processing is created (lines 9–11). The meaning of this timer is properly clarified in Section 5.6.4.

### 5.6.2 Message Handling

The main part of the protocol is the message handling code. The method invoked when a message is received is `handleGroupSendableEvent(...)` (see Listing 5.2) which is divided in two parts: dispatching outgoing messages (lines 2–26) and processing of incoming messages (lines 27–37).

We start describing the first part. The local sequence number is appended to every outgoing message (line 3) along with a flag stating if it is the first message (lines 5–11). If the protocol is in the middle of a switchover then a copy of the message must be sent using the second channel (lines 13–19). Finally, the message is directed to the proper total order algorithm by setting the associated channel (lines 20–23), and the local sequence number is incremented (line 25).

The second part starts by extracting the flag that identifies the first message (line 28). Then a condition is fired that will select the method that should process the message according to the channel where it was

received (lines 29–32). After the message is processed, if the termination condition is reached, the switchover ends (lines 34–35).

```
1  private void handleGroupSendableEvent(GroupSendableEvent event) {
2    if (event.getDir() == Direction.DOWN) {
3      event.getMessage().pushLong(localSN);
4
5      if (switching && isFirstMsg && !nullFirst) {
6        event.getMessage().pushBoolean(true);
7        isFirstMsg = false;
8        msgFirst = true;
9      }
10     else
11       event.getMessage().pushBoolean(false);
12
13     if (switching) {
14       GroupSendableEvent clone = (GroupSendableEvent) event.cloneEvent();
15       clone.setChannel(otherChannel);
16       clone.setSource(this);
17       clone.init();
18       clone.go();
19     }
20     event.setChannel(currentChannel);
21     event.setSource(this);
22     event.init();
23     event.go();
24
25     localSN++;
26   }
27   else { // DIRECTION UP
28     boolean flag = event.getMessage().popBoolean();
29     if (event.getChannel() == currentChannel)
30       processCurrent(event, flag);
31     else if (event.getChannel() == otherChannel)
32       processOther(event);
33
34     if (switching && checked >= actives)
35       endSwitch();
36     }
37   }
38 }
```

Listing 5.2: Main loop.

### 5.6.3 Message Processing

The incoming message processing is performed in two distinct methods, one for the messages received in the channel that contains the current total

```
1  private void processCurrent(GroupSendableEvent event, boolean flag) {
2    EventContainer cont = new EventContainer(event.orig,
3      event.getMessage().peekLong(),null);
4    if (switching && flag) {
5      check[event.orig] = true;
6      checked++;
7    }
8
9    tryDeliver(event, flag);
10
11   if (otherList.contains(cont))
12     otherList.remove(cont);
13 }
```

Listing 5.3: Message processing (current TO algorithm).

```
1  private void processOther(GroupSendableEvent event) {
2    EventContainer cont = new EventContainer(event.orig,
3      event.getMessage().popLong(),event);
4
5    if (cont.sn > lastDelivered[event.orig])
6      otherList.add(cont);
7  }
```

Listing 5.4: Message processing (next TO algorithm).

order protocol (`processCurrent(...)`), and the other for the ones received in the channel containing the next total order protocol (`processOther(...)`).

The first method (see Listing 5.3) starts by checking if the message is flagged, and increasing an internal counter accordingly (lines 4–7). It then delivers the message to the application (line 9) and removes any buffered copy of the message that might have been previously stored (lines 11–12), in case the message was first received in the other channel.

The second method (see Listing 5.4) simply checks if the message has already been delivered by the first method, otherwise it stores the message in a buffer for future delivery (lines 5–6).

```
1  private void handleNullEventTimer(NullEventTimer timer) {
2    if (switching && !msgFirst) {
3      NullEvent nullEventA = new NullEvent(currentChannel, Direction.DOWN, this,
4        vs.group, vs.id);
5      nullEventA.go();
6      nullFirst = true;
7    }
8  }
```

Listing 5.5: Null message production.

### 5.6.4 Null Message Handling

As the reader already knows, null messages are special messages used in the protocol to provide faster termination in situations where there are no application messages to be broadcast. In Section 5.6.1 we made reference to the creation of a timer that was associated with this kind of messages. When that timer expires, the method `handleNullTimer(...)` is invoked (see Listing 5.5). This method creates a null message and broadcasts it to the communication group (lines 3–5).

Every process in this group receives this message and invokes method `handleNullEvent(...)` (see Listing 5.6), which is responsible for processing this kind of messages. If a null message is received during the switching procedure, the internal counter associated with the origin of the message is updated (lines 2–4) and the termination condition is tested and may produce the protocol finalization (lines 6–7).

### 5.6.5 Buffer Cleanup

During the finalization of the protocol, a sanity procedure is required to clean up the messages that were kept buffered, because they were received in the second channel, rather than in the one currently in use. This procedure is implemented by the `cleanBuffers()` method (see Listing 5.7). The

```
1  private void handleNullEvent(NullEvent event) {
2    if (switching) {
3      check[event.orig] = true;
4      checked++;
5
6      if (checked == vs.view.length)
7        endSwitch();
8    }
9  }
```

Listing 5.6: Null message handling.

```
1  private void cleanBuffers() {
2    Iterator it = otherList.iterator();
3    while (it.hasNext()) {
4      EventContainer cont = (EventContainer) it.next();
5      cont.event.setChannel(applChannel);
6      cont.event.setSource(this);
7      cont.event.init();
8      cont.event.go();
9      lastDelivered[cont.source] = cont.sn;
10   }
11 }
```

Listing 5.7: Buffer cleanup.

list of stored (undelivered) messages is traversed and each message is delivered to the application in the proper order (lines 3–10).

### 5.6.6 Termination

When the termination condition is reached during the switchover, the finalization method is invoked (see Listing 5.8). This method works by switching over the references to the currently selected channel and the secondary channel (lines 3-10). After this the switching is terminated (line 12) and the protocol continues executing as normally.

```
1  private void endSwitch() {
2    cleanBuffers();
3    if (currentChannel == defaultChannel) {
4      currentChannel = secondChannel;
5      otherChannel = defaultChannel;
6    }
7    else {
8      currentChannel = defaultChannel;
9      otherChannel = secondChannel;
10   }
11
12   switching = false;
13 }
```

Listing 5.8: Protocol termination.

## 5.7 Summary

In this chapter we described an implementation of the proposed algorithm in the *Appia* protocol composition and execution framework that served the purpose of validating the functional properties of the algorithm and as the subject of the evaluation experiments described in the next chapter.

# Chapter 6

# Evaluation and Optimization

To evaluate the performance of our switching protocol we devised an experimental environment where we performed several tests. This chapter describes this environment and provides all the results obtained, along with a detailed discussion. In the end we present an optimization that allows the protocol to overcome some of the limitations identified in the experiments.

## 6.1 Experimental Environment

All the experiments described were conducted in the SSFNet (Nicol *et al.*, 2003) network simulator. A single network scenario was used and consists of a five node cluster, where all nodes are connected to each other by 100Mbps bi-directional links, as depicted in Figure 6.1.

The implementation of the adaptive protocol used is the one described in Chapter 5, which uses the *Appia* protocol composition and execution framework.

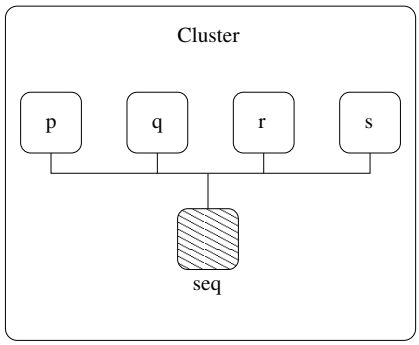


Figure 6.1: Simulation scenario.

## 6.2 Performance Evaluation

We evaluate the performance of our adaptive protocol from two different perspectives. First, we evaluate the overhead of the switching procedure. Then, we provide a comparative analysis on how different switching strategies interfere with the traffic flow during the reconfiguration.

### 6.2.1 Switching Overhead

To evaluate the switching overhead of our adaptive protocol we compare the performance of a system that always uses the same total order algorithm, with that of a system that is periodically switching between two algorithms. To make the comparison as fair as possible, we made our protocol switch between two instances of the same total order algorithm, which is also used as the non-adaptive protocol. Also, the network topology and working conditions did not change during the tests. In this way, we can isolate the cost of the switching procedure given that all the remaining factors remain unchanged.

Two runs of the same experiment were performed: (A) one using a single total order protocol (non-adaptive), (B) and another using the pro-

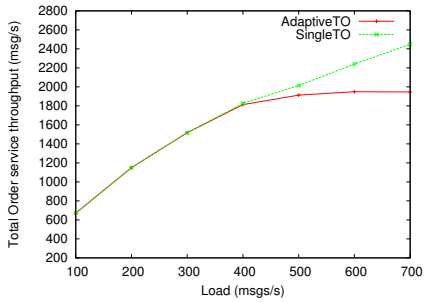


Figure 6.2: TO throughput in non-adaptive and adaptive algorithms

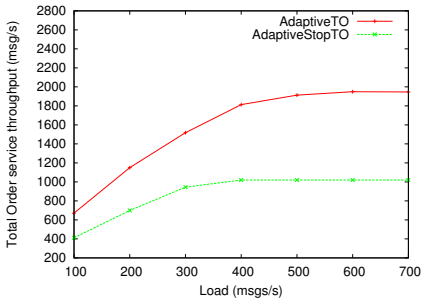


Figure 6.3: TO throughput in adaptive and stop algorithms

posed adaptive total order protocol, which is forced to switch periodically. Each run consists of every node broadcasting 5000 messages of 5KB in total order. The experiment ends when all nodes receive all the broadcast messages. The values presented are averages of the measurements conducted in each node.

Figure 6.2 presents the overall throughput results when the send rate is made variable. As depicted, both total order algorithms perform the same until they reach approximately 400 msg/s. After this point, the throughput of the non-adaptive protocol continues to grow while its value stabilizes for the adaptive protocol. This behavior is explained by the overhead introduced by the switching phase in the adaptive protocol. During this phase, the same set of messages is being broadcast by two total order algorithms at the same time, leading to an increase (approximately double) in the bandwidth usage. If the send rate is too high, the available bandwidth can be exhausted, leading to the stagnation observed in the throughput.

Thus, we can conclude that our switching protocol offers negligible overhead as long as there is enough network bandwidth to support the transmission of data in parallel during the reconfiguration. When the protocol operates close to the available bandwidth, the switching procedure

introduces an overhead. This limitation can be addressed at the implementation level, by sending the payload of the messages using just one of the two algorithms. This optimization is described in Section 6.3.

### 6.2.2 Comparative Analysis

Most switching protocols require the message flow to be stopped in order to terminate the reconfiguration process. By not imposing a gap in the message flow, our protocol provides smooth transitions between algorithms, thus allowing applications that rely in its services to normally execute, even during the switching phase. Therefore, it should offer better overall throughput, as long as enough bandwidth is available to cope with the demand imposed by the transmission of messages using two algorithms at the same time. The same experiment described in 6.2.1 was conducted using a protocol that stops the message flow. This protocol operates by sending a stop request to all nodes and awaiting for a confirmation from each of these nodes. After confirming the stop request a node does not send further messages until the switch is complete. The performance of such protocol when compared to our proposal can be observed in Figure 6.3, which clearly shows that our approach always performs better.

Other protocols that try to minimize the cost of switching between algorithms have also been proposed. A previous work (Rutti *et al.*, 2006), described in Section 2.4.4, proposes a solution that has some similarities with our protocol, but differs from it by not requiring every node to wait for a "special" (in our algorithm the term is "flagged") message from every other node, and also for not making any assumptions about the failure model where it is executing (see Section 4.3). In (Rutti *et al.*, 2006), a special

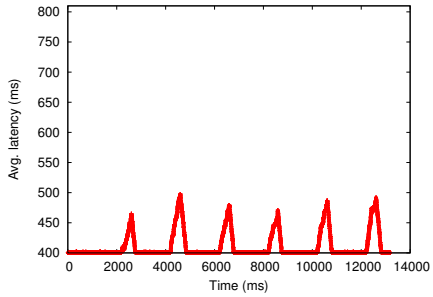


Figure 6.4: Latency in Adaptive TO

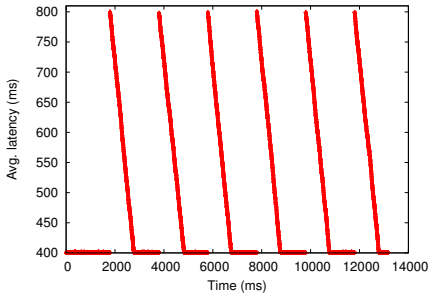


Figure 6.5: Latency in RABP

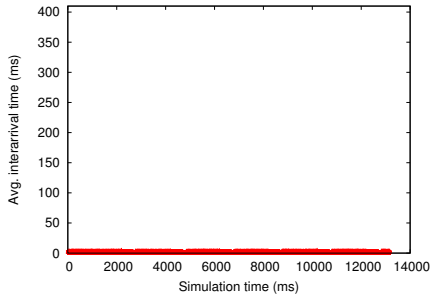


Figure 6.6: Inter-arrival time in Adaptive TO

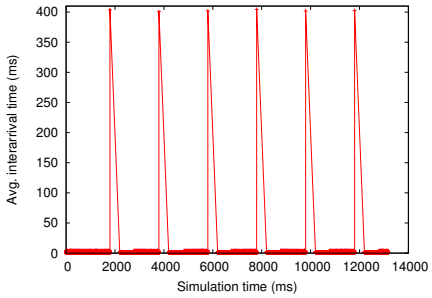


Figure 6.7: Inter-arrival time in RABP

reconfiguration message is broadcast in total order. When a node receives such message, it stops the flow in the current algorithm, and re-issues all his undelivered messages in the next algorithm. It then starts using it to broadcast messages in total order. We will refer to this protocol by RABP (*Replacement of the Atomic Broadcast Protocol*).

The RABP strategy has the advantage of requiring less bandwidth during the switching phase. However, some delay is imposed to the message flow during the retransmission of the undelivered messages. To observe this side effect, the experiment was now conducted using our protocol and the RABP protocol. In Figures 6.4 and 6.5 we can observe how both compare in terms of latency. The spikes depicted correspond to the switching phases, in the timeline of the experiment.

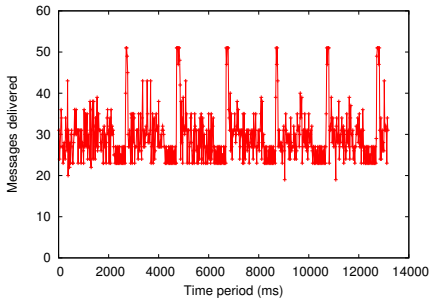


Figure 6.8: Delivery rate in Adaptive TO

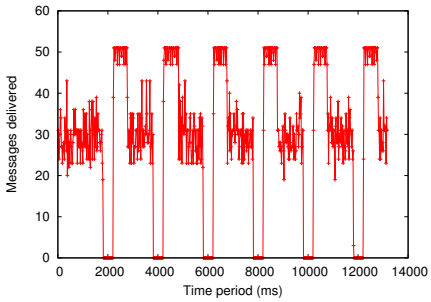


Figure 6.9: Delivery rate in RABP

The inter-arrival time of messages was also measured and its evolution is shown in Figures 6.6 and 6.7. Finally, the number of messages delivered by a fixed period of time (10 ms) was also observed and the comparative results are depicted in Figures 6.8 and 6.9.

This experiment clearly showed that our proposal is able to keep a sustained delivery rate during the switching phase and performs similarly to RABP during the remaining time. By not significantly delaying the message flow, our protocol can best suit environments where application stoppage, due to significant communication delays, is not desirable.

## 6.3 Implementation Optimization

When enough bandwidth is available, the (non-optimized) version of our protocol already implements the switching procedure with negligible overhead in the message flow. However, the experimental results provided in Section 6.2 showed that during the switching phase, when both protocols are being used to broadcast the same set of messages, the available bandwidth can be exhausted when the send rate and/or message payload is too high.

To overcome this problem we now describe an optimization to reduce

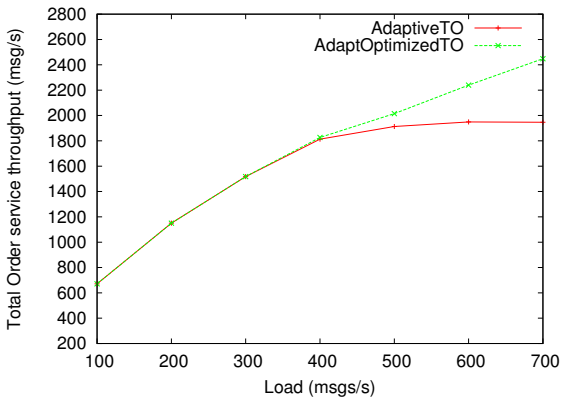


Figure 6.10: Total Order service throughput *vs* message send rate

the amount of data being transmitted by the adaptive protocol during this phase. The optimization consists of broadcasting using the first (and current) algorithm only the identifiers of the messages being transmitted. The messages payload is only transmitted using the second algorithm. In this manner, the amount of redundant information transmitted over the wire is reduced substantially. This optimization has a minor drawback: the protocol cannot deliver a message to the application until it is received by both total order algorithms. However, since both algorithms are executed in parallel, the impact of this feature is negligible.

### 6.3.1 Performance Evaluation

The same experiment as described in Section 6.2.1 was performed using the optimized implementation of the switching protocol. Figure 6.10 shows that the optimization allows the protocol to continue increasing its throughput after the point where the non-optimized version stabilizes (approximately 400 msg/s), showing a behavior similar to the non-adaptive protocol.

## 6.4 Summary

This chapter provided evaluation results of several experiments designed to assess the benefits of our proposal in respect to related approaches. The overall results from the evaluation experiments are whithin the expectations, however, some performance deterioration was identified due to shortages in the available bandwidth. To reduce this effect an optimization technique was proposed and a brief evaluation was presented.

# Chapter 7

# Conclusions

Total order broadcast protocols implement an important service required by several fault-tolerant distributed applications. Several different strategies that implement such service have been proposed, that may perform better in specific environments and/or working conditions.

The expensive nature of these protocols motivated for the development of optimistic alternatives, that rely on quite different assumptions about the execution environment, and allow the existence of early deliveries that are estimates of the final total order delivery of a common total order protocol.

This dissertation described the characteristics and assumptions of the currently available optimistic total order protocols, and established an abstract timeline for the different optimistic deliveries, that motivates the design and use of adaptive protocols.

We also presented an adaptive total order protocol that is able to switch in run-time between different total order algorithms (not just optimistic ones). When the environment is dynamic, this allows the system to use the ordering strategy that is most favorable.

If one is not careful, the procedure to switch between algorithms can disrupt the message flow. Our work tackles with this issue by proposing a novel switching strategy that performs the reconfiguration with negligible impact on the observed delivery rate. We have implemented and evaluated our protocol in isolation and compared it with competing approaches. The results show a negligible interference in the message flow as long as enough bandwidth is available to cope with the demand of the switching protocol. Even in scenarios where enough bandwidth is not available, a simple optimization is described that requires less bandwidth at the cost of a possible increase in the switching procedure duration.

## 7.1 Future Work

Part of the motivation for this research was the possibility of using adaptive total order algorithms to improve the performance of database replication services based on the state machine approach (Schneider, 1990). We plan to embed our adaptive algorithm in a database replication service being implemented in the context of the IST project GORDA (GORDA Consortium, 2005), and perform the required performance analysis to assess the benefits of using such approach.

Also related to database replication services, the study of the optimal switching point where reconfiguration should be performed amongst different optimistic total order protocols is a planned objective.

# Bibliography


BIRMAN, K., & JOSEPH, T. 1987a (Feb.). *Exploiting Virtual Synchrony in Distributed Systems*. Tech. rept. 87-811. Department of Computer Science, Cornell University, Ithaca, New York.

BIRMAN, K., & JOSEPH, T. 1987b. Reliable communication in the presence of failures. *ACM, Transactions on Computer Systems*, **5**(1).

CHANDRA, T., & TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, **43**(2), 225–267.

CHANG, J., & MAXEMCHUCK, N. 1984. Reliable broadcast protocols. *ACM, Transactions on Computer Systems*, **2**(3).

CHEN, W.-K., HILTUNEN, M., & SCHLICHTING, R. 2001. Constructing Adaptive Software in Distributed Systems. *Page 635 of: ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society.

DÉFAGO, X., SCHIPER, A., & URBÁN, P. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, **36**(4), 372–421.

DOLEV, D., KRAMER, S., & MALKI, D. 1993. Early delivery totally ordered multicast in asynchronous environments. *Pages 544–553 of: Digest*

*of Papers, The 23th International Symposium on Fault-Tolerant Computing*. IEEE.

GORDA CONSORTIUM. 2005 (Mar.). *GORDA Architecture Definition*. GORDA Deliverable 2.2.

GUERRAOUI, R., & SCHIPER, A. 1997. Software-Based Replication for Fault Tolerance. *IEEE Computer*, **30**(4), 68–74.

HAYDEN, M. 1998. *The Ensemble System*. Ph.D. thesis, Cornell University, Computer Science Department.

HILTUNEN, M., & SCHLICHTING, R. 2000 (Oct.). The Cactus approach to building configurable middleware services. *In: Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*.

KAASHOEK, M., & TANENBAUM, A. 1991. Group communication in the Amoeba distributed operating system. *Pages 222–230 of: Proceedings of the 11th International Conference on Distributed Computing Systems*. IEEE.

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

LIU, X., & VAN RENESSE, R. 2000 (July). Fast Protocol Transition in A Distributed Environment. *Page 341 of: Proceedings of the 19th ACM Conference on Principles of Distributed Computing (PODC 2000)*.

MIRANDA, H., PINTO, A., & RODRIGUES, L. 2001. Appia, a flexible protocol kernel supporting multiple coordinated channels. *Pages 707–710 of: Proceedings of the 21st International Conference on Distributed Computing Systems*. Phoenix, Arizona: IEEE.

MOCITO, J., & RODRIGUES, L. 2006. Run-Time Switching Between Total Order Algorithms. *In: Proceedings of the Euro-Par 2006*. LNCS. Dresden, Germany: Springer-Verlag.

NICOL, D., LIU, J., LILJENSTAM, M., & YAN, G. 2003. Simulation of Large-Scale Networks Using SSF. *In: Proceedings of the 2003 Winter Simulation Conference*.

PEDONE, F., & SCHIPER, A. 1998. Optimistic Atomic Broadcast. *Pages 318–332 of: Proceedings of the 12th International Symposium on Distributed Computing (DISC'98)*. London, UK: Springer-Verlag.

PETERSON, L., BUCHHOLZ, N., & SCHLICHTING, R. 1989. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, **7**(3), 217–146.

POWELL, D. (ed). 1996. *Communications of the ACM*. Vol. 39. ACM. Chap. Special Issue on Group Communication, pages 50–97.

RODRIGUES, L., FONSECA, H., & VERÍSSIMO, P. 1996 (May). Totally Ordered Multicast in Large-Scale Systems. *Pages 503–510 of: Proceedings of the 16th International Conference on Distributed Computing Systems*. IEEE, Hong Kong.

RODRIGUES, L., MOCITO, J., & CARVALHO, N. 2006. From Spontaneous Total Order to Uniform Total Order: different degrees of optimistic delivery. *In: Proceedings of the 21st ACM symposium on Applied Computing (SAC'06)*. ACM Press.

RUTTI, O., WOJCIECHOWSKI, P., & SCHIPER, A. 2006. Structural and Algorithmic Issues of Dynamic Protocol Update. *In: Proceedings of*

*the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, **22**(4), 299–319.

SOUSA, A., PEREIRA, J., MOURA, F., & OLIVEIRA, R. 2002. Optimistic Total Order in Wide Area Networks. *Pages 190–199 of: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*. Washington, DC, USA: IEEE Computer Society.

VAN RENESSE, R., BIRMAN, K., HAYDEN, M., VAYSBURD, A., & KARR, D. 1998. Building adaptive systems using Ensemble. *Software: Practice and Experience*, **28**(9), 963–979.